PYTHON 基础教程

人生苦短，我用 PYTHON

版本：2019.3.20

作者：陈福明

目录

第一章 Python 基础知识

——认识 Python

人生苦短，我用 Python —— Life is short, you need Python

图1-1 生苦短我用python

目标

- Python 的起源
- 为什么要用 Python?
- Python 的特点
- Python 的优缺点

1. Python 的起源

Python 的创始人为吉多·范罗苏姆（Guido van Rossum）

图1-2 吉多

1. 1989 年的圣诞节期间，吉多·范罗苏姆为了在阿姆斯特丹打发时间，决心开发一个新的**解释程序**，作为 ABC 语言的一种继承（**感觉下什么叫牛人**）
2. ABC 是由吉多参加设计的一种教学语言，就吉多本人看来，ABC 这种语言非常优美和强大，是**专门为非专业程序员设计的**。但是 ABC 语言并没有成功，究其原因，吉多认为是**非开放**造成的。吉多决心在 Python 中避免这一错误，并获取了非常好的效果
3. 之所以选中 Python（蟒蛇）作为程序的名字，是因为他是 BBC 电视剧——蒙提·派森的飞行马戏团（Monty Python's Flying Circus）的爱好者
4. 1991 年，第一个 Python **解释器** 诞生，它是用 C 语言实现的，并能够调用 C 语言的库文件

1.1 解释器（科普）

计算机不能直接理解任何除机器语言以外的语言，所以必须要把程序员所写的程序语言翻译成机器语言，计算机才能执行程序。**将其他语言翻译成机器语言的工具，被称为编译器**

编译器翻译的方式有两种：一个是**编译**，另外一个是**解释**。两种方式之间的区别在于**翻译时间点的不同**。当编译器**以解释方式运行的时候**，也称之为**解释器**

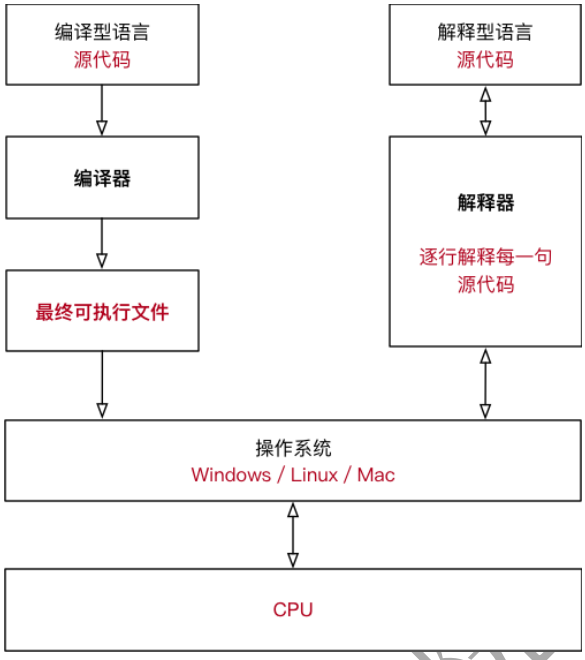

图1-3 编译型和解释型语言工作对比

- **编译型语言**：程序在执行之前需要一个专门的编译过程，把程序编译成为机器语言的文件，运行时不需要重新翻译，直接使用编译的结果就行了。程序执行效率高，依赖编译器，跨平台性差些。如 C、C++
- **解释型语言**：解释型语言编写的程序不进行预先编译，以文本方式存储程序代码，会将代码一句一句直接运行。在发布程序时，看起来省了道编译工序，但是在运行程序的时候，必须先解释再运行

编译型语言和解释型语言对比

- **速度** —— 编译型语言比解释型语言执行速度快
- **跨平台性** —— 解释型语言比编译型语言跨平台性好

1.2 Python 的设计目标

1999 年，吉多·范罗苏姆向 DARPA 提交了一条名为“Computer Programming for Everybody”的资金申请，并在后来说明了他对 Python 的目标：

- 一门**简单直观的语言**并与主要竞争者一样强大
- **开源**，以便任何人都可以为它做贡献
- 代码**像纯英语那样容易理解**
- 适用于**短期**开发的日常任务

这些想法中的基本都已经成为现实，Python 已经成为一门流行的编程语言

1.3 Python 的设计哲学

1. 优雅
2. 明确
3. 简单

- Python 开发者的哲学是：**用一种方法，最好是只有一种方法来做一件事**
- 如果面临多种选择，Python 开发者一般会拒绝花俏的语法，而选择**明确没有或者很少有歧义的语法**

在 Python 社区，吉多被称为“仁慈的独裁者”

2. 为什么选择 Python?

- 代码量少
- ……

同一样问题，用不同的语言解决，代码量差距还是很多的，一般情况下 `Python` 是 `Java` 的 **1/5**，所以说 **人生苦短，我用 Python**

3. Python 特点

- Python 具有通用性。

Python 语言可以用于几乎任何与程序设计相关 应用的开发，不仅适合训练变成思维，更适合诸如 数据分析、机器学习、人工智能、Web 开发等具体的 技术领域。

- Python 语法简洁。

Python 语法主要用来精确表达问题逻辑，更接近自然语言，只有 33 个保留字，十分简洁。

- Python 生态高产。

Python 解释器提供了几百个内置类和函数库， 此外，世界各地程序员通过开源社区贡献了十几万 个第三方函数库，几乎覆盖了计算机技术的各个领 域，编写 Python 程序可以大量利用已有内置或第三 方代码，具备良好的编程生态。

除了 Python 语法的三个重要特点外，Python 程序还有一些具体特点。

  - 平台无关
  - 强制可读
  - 支持中文

- 平台无关

Python 程序可以在任何安装解释器的计算机环 境中执行，因此，可以不经修改地实现跨操作系统 运行。

- 强制可读

Python 通过强制缩进（类似文章段落的首行空 格）来体现语句间的逻辑关系，显著提高了程序的 可读性，进而增强了 Python 程序的可维护性。

- 支持中文

Python 3.x 版本采用 Unicode 编码表达所有字 符信息。Unicode 是一种国际通用表达字符的编码体 系，这使得 Python 程序可以直接支持英文、中文、 法文、德文等各类自然语言字符，在处理中文时更 加灵活且高效。

- Python 是**完全面向对象的语言**
  - **函数**、**模块**、**数字**、**字符串**都是对象，**在 Python 中一切皆对象**
  - 完全支持继承、重载、多重继承
  - 支持重载运算符，也支持泛型设计
- Python **拥有一个强大的标准库**，Python 语言的核心只包含 **数字**、**字符串**、**列表**、**字典**、**文件** 等常见类型和函数，而由 Python 标准库提供了 **系统管理**、**网络通信**、**文本处理**、**数据库接口**、**图形系统**、**XML 处理** 等额外的功能
- Python 社区提供了**大量的第三方模块**，使用方式与标准库类似。它们的功能覆盖 **科学计算**、**人工智能**、**机器学习**、**Web 开发**、**数据库接口**、**图形系统** 多个领域

4. Python 的优缺点

4.1 优点

- 简单、易学
- 免费、开源
- **面向对象**
- 丰富的库
- 可扩展性
  - 如果需要一段关键代码运行得更快或者希望某些算法不公开，可以把这部分程序用 `C` 或 `C++` 编写，然后在 `Python` 程序中使用它们
- ……

4.2 缺点

- 运行速度
- 移动开发很少
- 版本兼容性不好
- ……

第二章 第一个 Python 程序

学习目标

- 第一个 Hello Python 程序
- Python 2.x 与 3.x 版本简介
- 执行 Python 程序的三种方式
  - 解释器 —— python / python3
  - 交互式 —— ipython
  - 简单的开发环境——IDLE
  - 集成开发环境 —— PyCharm

1. 第一个 Hello Python 程序

1.1 Python 源程序的基本概念

1. Python 源程序就是**一个特殊格式的文本文件**，可以**使用任意文本编辑软件**做 Python 的开发
2. Python 程序的 **文件扩展名** 通常都是 .py

1.2 演练步骤

- 在桌面下，新建 认识 Python 目录
- 在 认识 Python 目录下新建 hellopython.py 文件
- 使用记事本等文本编辑器编辑 hellopython.py 并且输入以下内容：

```
print("hello world ")
print("你好！！ ")
```

- 在终端中输入以下命令执行 `hellopython.py`

```
$ python hellopython.py
C:\> python hellopython.py
```

`print` 是 `python` 中我们学习的第一个 **函数**

`print` 函数的作用，可以把 "" 内部的内容，输出到屏幕上

1.3 演练扩展 —— 认识错误（BUG）

关于错误

- 编写的程序**不能正常执行**，或者**执行的结果不是我们期望的**
- 俗称 `BUG`，是程序员在开发时非常常见的，初学者常见错误的原因包括：
  1. 手误
  2. 对已经学习过的知识理解还存在不足
  3. 对语言还有需要学习和提升的内容
- 在学习语言时，不仅要**学会语言的语法**，而且还要**学会如何认识错误和解决错误的方法**

每一个程序员都是在不断地修改错误中成长的

第一个演练中的常见错误

- 1> **手误**，例如使用 `pirnt("Hello world")`

```
NameError: name 'pirnt' is not defined
```

名称错误：'pirnt' 名字没有定义

- 2> 将多条 `print` 写在一行

```
SyntaxError: invalid syntax
```

语法错误：语法无效

每行代码负责完成一个动作

- 3> 缩进错误

```
IndentationError: unexpected indent
```

缩进错误：不期望出现的缩进

- Python 是一个格式非常严格的程序设计语言
- 目前而言，大家记住每行代码前面都不要增加空格

- 4> **python 2.x 默认不支持中文**

目前市场上有两个 Python 的版本并存着，分别是 `Python 2.x` 和 `Python 3.x`

- **Python 2.x 默认不支持中文**，具体原因，等到介绍 **字符编码** 时给大家讲解
- Python 2.x 的解释器名称是 **python**
- Python 3.x 的解释器名称是 **python**

```
SyntaxError: Non-ASCII character '\xe4' in file 01-HelloPyth
on.py on line 3,

but no encoding declared;

see http://python.org/dev/peps/pep-0263/ for details
```

语法错误： 在 `hellopython.py` 中第 `3` 行出现了非 `ASCII` 字符 `'\xe4'`，但是没有声明文件编码

请访问 `http://python.org/dev/peps/pep-0263/` 了解详细信息

- `ASCII` 字符只包含 `256` 个字符，不支持中文
- 有关字符编码的问题，后续会讲

单词列表

```
* error 错误

* name 名字

* defined 已经定义

* syntax 语法

* invalid 无效

* Indentation 索引

* unexpected 意外的，不期望的
```

* character 字符

* line 行

* encoding 编码

* declared 声明

* details 细节，详细信息

* ASCII 一种字符编码

2. Python 2.x 与 3.x 版本简介

目前市场上有两个 Python 的版本并存着，分别是 Python 2.x 和 Python 3.x

新的 Python 程序建议使用 Python 3.0 版本的语法

- Python 2.x 是 **过去的版本**
  - 解释器名称是 **python2**
- Python 3.x 是 **现在和未来 主流的版本**
  - 解释器名称是 **python**
  - 相对于 Python 的早期版本，这是一个 **较大的升级**
  - 为了不带入过多的累赘，Python 3.0 在设计的时候 **没有考虑向下兼容**
    - 许多早期 Python 版本设计的程序都无法在 Python 3.0 上正常执行
  - Python 3.0 发布于 **2008 年**
  - 到目前为止，Python 3.0 的稳定版本已经有很多年了
    - Python 3.3 发布于 2012
    - Python 3.4 发布于 2014
    - Python 3.5 发布于 2015
    - Python 3.6 发布于 2016
    - Python 3.7 发布于 2018
- 为了照顾现有的程序，官方提供了一个过渡版本 —— **Python 2.6**
  - 基本使用了 Python 2.x 的语法和库
  - 同时考虑了向 Python 3.0 的迁移，**允许使用部分** Python 3.0 的语法与函数
  - 2010 年中推出的 Python 2.7 被确定为 **最后一个 Python 2.x 版本**

提示：如果开发时，无法立即使用 Python 3.0（还有极少的第三方库不支持 3.0 的语法），建议

- 先使用 Python 3.0 版本进行开发
- 然后使用 Python 2.6、Python 2.7 来执行，并且做一些兼容性的处理

3. 执行 Python 程序的三种方式

3.1. 解释器 python2/ python3

Python 的解释器

```
# 使用 python 2.x 解释器
$ python2 xxx.py
C:\ > python2 xxx.py
# 使用 python 3.x 解释器
$ python xxx.py
C:\ > python xxx.py
```

其他解释器（知道）

Python 的解释器 如今有多个语言的实现，包括：

- CPython —— 官方版本的 C 语言实现
- Jython —— 可以运行在 Java 平台
- IronPython —— 可以运行在 .NET 和 Mono 平台
- PyPy —— Python 实现的，支持 JIT 即时编译

3.2. 交互式运行 Python 程序

- 直接在终端中运行解释器，而不输入要执行的文件名
- 在 Python 的 Shell 中直接输入 **Python 的代码**，会立即看到程序执行结果

1) 交互式运行 Python 的优缺点

优点

- 适合于学习/验证 Python 语法或者局部代码

缺点

- 代码不能保存
- 不适合运行太大的程序

2) 退出 官方的解释器

1> 直接输入 exit()

```
>>> exit()
```

2> 使用热键退出

在 python 解释器中，按热键 ctrl + d 可以退出解释器

3) IPython

- IPython 中 的 “I” 代表 **交互 interactive**

特点

- IPython 是一个 python 的 **交互式 shell**，比默认的 python shell 好用得多
  - 支持自动补全
  - 自动缩进
  - 支持 bash shell 命令
  - 内置了许多很有用的功能和函数
- IPython 是基于 BSD 开源的

版本

- Python 2.x 使用的解释器是 **ipython2**
- Python 3.x 使用的解释器是 **ipython3**
- 要退出解释器可以有以下两种方式：

1> 直接输入 exit

```
In [1]: exit
```

2> 使用热键退出

在 IPython 解释器中，按热键 ctrl + d，IPython 会询问是否退出解释器

IPython 的安装

```
$ sudo apt install ipython
C:\ >pip install ipython
```

3.3. Python 的 IDE —— IDLE

1）集成开发环境（IDE）

集成开发环境（IDE，Integrated Development Environment）—— **集成了开发软件需要的所有工具**，一般包括以下工具：

- 图形用户界面
- 代码编辑器（支持 **代码补全 / 自动缩进**）
- 编译器 / 解释器
- 调试器（**断点 / 单步执行**）
- ……

2）IDLE 介绍

- IDLE 是 Python 自带的一款简洁的集成开发环境
- IDLE 除了具有一般 IDE 所必备功能外，还可以在 Windows、Linux、macOS 下使用
- IDLE 适合开发中小型项目
    - 一个项目通常会包含 **很多源文件**
    - 每个 **源文件** 的代码行数是有限的，通常在几百行之内
    - 每个 **源文件** 各司其职，共同完成复杂的业务功能

python 安装后，默认自带此工具，启用：开始->程序->Python 2.*/3.*-> IDLE (Python GUI)如此就打开了 Python Shell，可以输入语句命令进行交互练习：

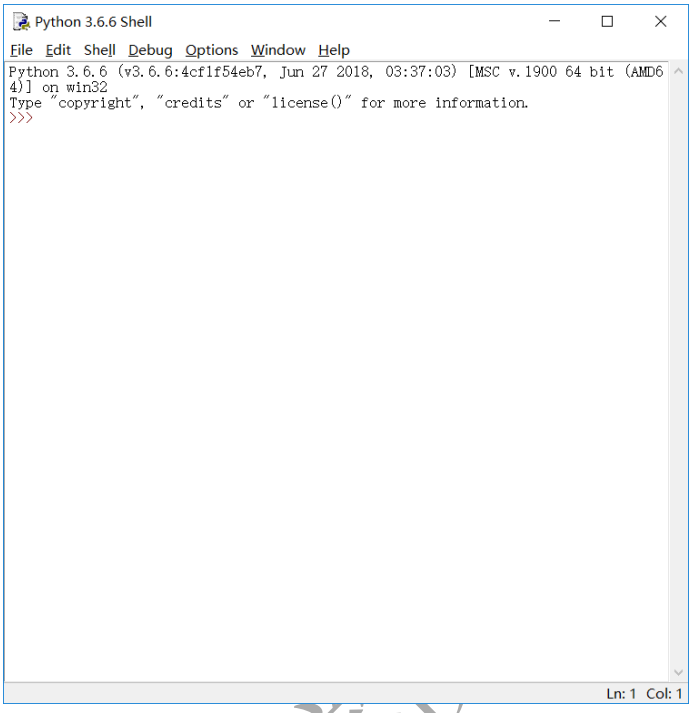

图 2-1 IDLE 的 Python Shell

3）IDLE 快速体验

IDLE 的 Python Shell 的菜单 File->New window(Ctrl+N)可以打开 Python 文件（右击任何一个.py 文件，弹出菜单中的“Edit with IDLE”也可以调用 IDLE 打开这个.py 文件然后进行调试）。

- **文件(File)菜单** 能够 **新建 / 定位 / 打开** Python 文件
- **文件编辑区域** 能够 **编辑** 当前打开的文件
- **运行(Run)菜单** 能够 **执行(F5)** 代码

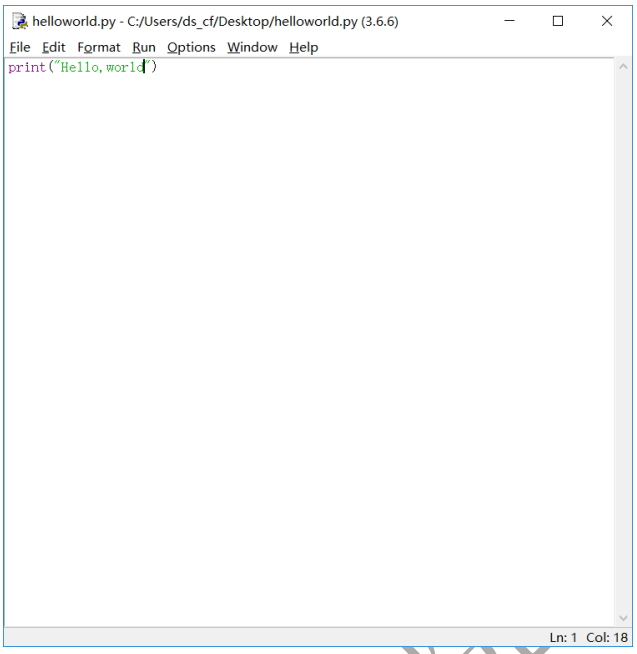

图 2-2 IDLE 编辑器

3.4. Python 的 IDE —— PyCharm

1）PyCharm 介绍

- PyCharm 是 Python 的一款非常优秀的集成开发环境
- PyCharm 除了具有一般 IDE 所必备功能外，还可以在 Windows、Linux、macOS 下使用
- PyCharm 适合开发大型项目
    - 一个项目通常会包含 **很多源文件**
    - 每个 **源文件** 的代码行数是有限的，通常在几百行之内
    - 每个 **源文件** 各司其职，共同完成复杂的业务功能

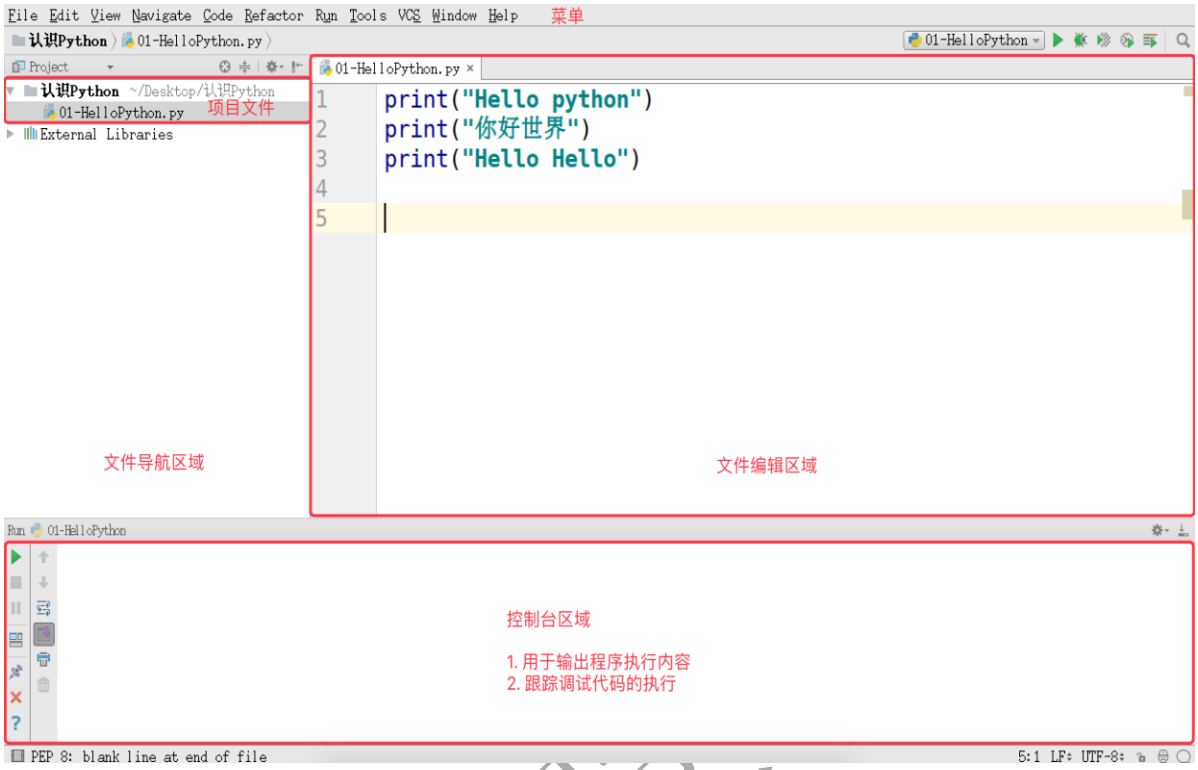

图 2-3 PyCharm 的界面结构

2）PyCharm 快速体验

- **文件导航区域** 能够 **浏览 / 定位 / 打开** 项目文件
- **文件编辑区域** 能够 **编辑** 当前打开的文件
- **控制台区域** 能够:
  - 输出程序执行内容
  - 跟踪调试代码的执行
- 右上角的 **工具栏** 能够 **执行(SHIFT + F10) / 调试(SHIFT + F9)** 代码

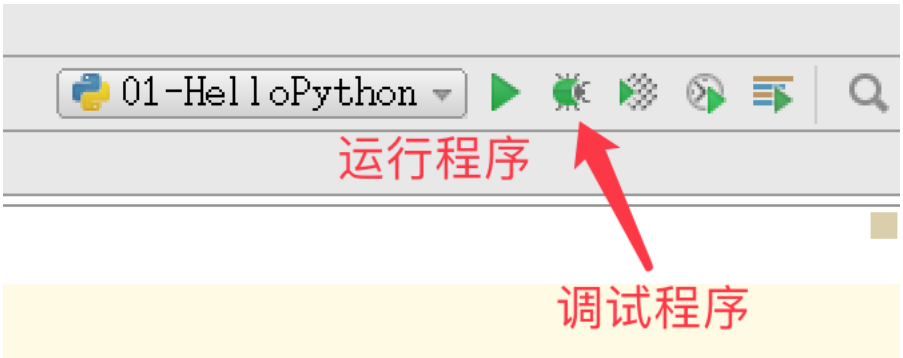

图 2-4 PyCharm 运行工具栏

- 通过控制台上方的**单步执行按钮(F8)**，可以单步执行代码

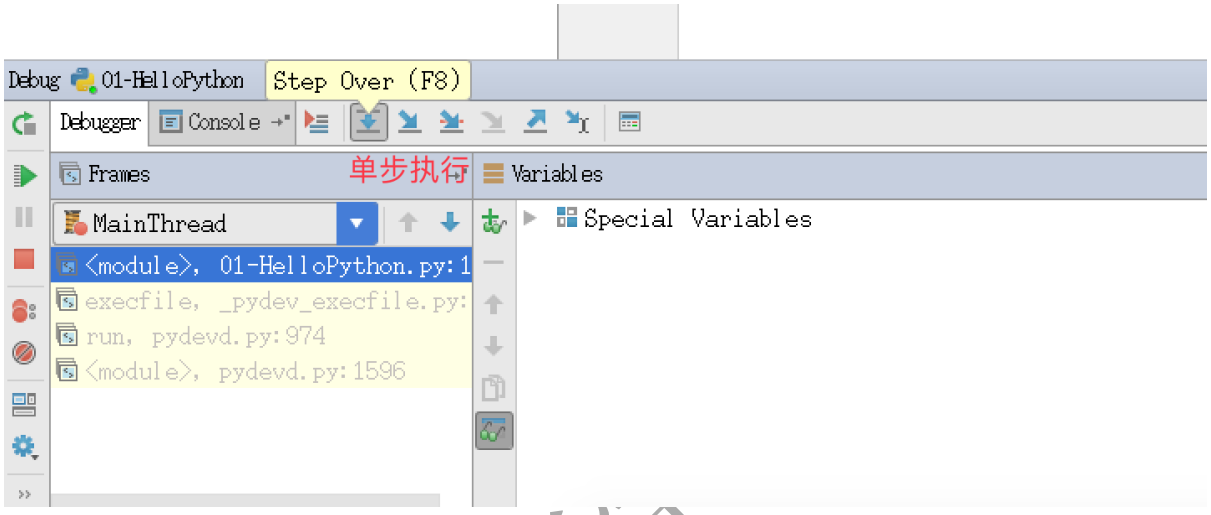

图 2-5 PyCharm 调试器

第三章 注释与变量

1. 注释

目标

- 注释的作用
- 单行注释（行注释）
- 多行注释（块注释）

1.1. 注释的作用

使用用自己熟悉的语言，在程序中对某些代码进行标注说明，增强程序的可读性

1.2. 单行注释(行注释)

- 以 `#` 开头，`#` 右边的所有东西都被当做说明文字，而不是真正要执行的程序，只起到辅助说明作用
- 示例代码如下：

```
# 这是第一个单行注释
print("hello python")
```

为了保证代码的可读性，`#` 后面建议先添加一个空格，然后再编写相应的说明文字

在代码后面增加的单行注释

- 在程序开发时，同样可以使用 `#` 在代码的后面（旁边）增加说明性的文字
- 但是，需要注意的是，**为了保证代码的可读性，注释和代码之间** 至少要有 **两个空格**
- 示例代码如下：

```
print("hello python")    # 输出 `hello python`
```

1.3. 多行注释（块注释）

- 如果希望编写的 **注释信息很多，一行无法显示**，就可以使用多行注释
- 要在 Python 程序中使用多行注释，可以用 **一对 连续的 三个 引号**(单引号和双引号都可以)
- 示例代码如下：

```
"""
这是一个多行注释
在多行注释之间，可以写很多很多的内容……
"""
print("hello python")
```

什么时候需要使用注释？

1. **注释不是越多越好**，对于一目了然的代码，不需要添加注释
2. 对于 **复杂的操作**，应该在操作开始前写上若干行注释
3. 对于 **不是一目了然的代码**，应在其行尾添加注释（为了提高可读性，注释应该至少离开代码 2 个空格）
4. 绝不要描述代码，假设阅读代码的人比你更懂 Python，他只是不知道你的代码要做什么

在一些正规的开发团队，通常会有 **代码审核** 的惯例，就是一个团队中彼此阅读对方的代码

关于代码规范

- `Python` 官方提供有一系列 PEP（Python Enhancement Proposals） 文档
- 其中第 8 篇文档专门针对 **Python 的代码格式** 给出了建议，也就是俗称的 **PEP 8**
- 文档地址：https://www.python.org/dev/peps/pep-0008/
- 谷歌有对应的中文文档：http://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/python_style_rules/

任何语言的程序员，编写出符合规范的代码，是开始程序生涯的第一步

2. 变量的命名

目标

- 标识符和关键字
- 变量的命名规则

2.1 标识符和关键字

2.1.1 标识符

标示符就是程序员定义的 **变量名、函数名**

名字 需要有 **见名知义** 的效果

- 标示符可以由 **字母**、**下划线** 和 **数字** 组成
- **不能以数字开头**
- **不能与关键字重名**

思考：下面的标示符哪些是正确的，哪些不正确为什么？

```
fromNo12

from#12

my_Boolean

my-Boolean

Obj2

2ndObj

myInt

My_tExt

_test

test!32
```

```
haha(da)tt

jack_rose

jack&rose

GUI

G.U.I
```

2.1.2 关键字

- **关键字** 就是在 `Python` 内部已经使用的标识符
- **关键字** 具有特殊的功能和含义
- 开发者 **不允许定义和关键字相同的名字的标示符**

通过以下命令可以查看 `Python` 中的关键字

```
In [1]: import keyword

In [2]: print(keyword.kwlist)
```

提示：**关键字的学习及使用**，会在后面的课程中不断介绍

- `import` **关键字** 可以导入一个 **“工具包”**
- 在 `Python` 中不同的工具包，提供有不同的工具

2.2. 变量的命名规则

命名规则 可以被视为一种 **惯例**，并无绝对与强制 目的是为了 **增加代码的识别和可读性**

注意 `Python` 中的 **标识符** 是 **区分大小写的**

1. 在定义变量时，为了保证代码格式，`=` 的左右应该各保留一个空格
2. 在 `Python` 中，如果 **变量名** 需要由 **二个** 或 **多个单词** 组成时，可以按照以下方式命名
    A. 每个单词都使用小写字母
    B. 单词与单词之间使用 **_下划线** 连接
    C. 例如：`first_name`、`last_name`、`qq_number`、`qq_password`

2.2.1 驼峰命名法

- 当 **变量名** 是由二个或多个单词组成时，还可以利用驼峰命名法来命名

- **小驼峰式命名法**

    1) 第一个单词以小写字母开始，后续单词的首字母大写
    2) 例如：`firstName`、`lastName`

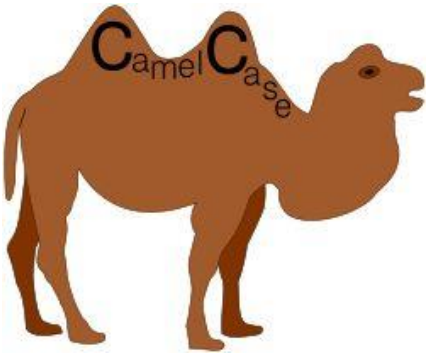

图 3-1 驼峰命名法

- **大驼峰式命名法**

    1) 每一个单词的首字母都采用大写字母
    2) 例如：`FirstName`、`LastName`、`CamelCase`

3. 变量的使用

程序就是用来处理数据的，而变量就是用来存储数据的

目标

- 变量定义
- 变量的类型
- 变量的命名

3.1. 变量定义

- 在 Python 中，每个变量 **在使用前都必须赋值**，变量 **赋值以后** 该变量 **才会被创建**
- 等号（=）用来给变量赋值

- = 左边是一个变量名
- = 右边是存储在变量中的值

```
变量名 = 值
```

变量定义之后，后续就可以直接使用了

3.1.1 变量演练 1 —— iPython

```
# 定义 qq_number 的变量用来保存 qq 号码
In [1]: qq_number = "1234567"

# 输出 qq_number 中保存的内容
In [2]: qq_number
Out[2]: '1234567'

# 定义 qq_password 的变量用来保存 qq 密码
In [3]: qq_password = "123"

# 输出 qq_password 中保存的内容
In [4]: qq_password
Out[4]: '123'
```

使用交互式方式，如果要查看变量内容，直接输入变量名即可，不需要使用 print 函数

3.1.2 变量演练 2 —— PyCharm

```
# 定义 qq 号码变量
qq_number = "1234567"
# 定义 qq 密码变量
qq_password = "123"
# 在程序中，如果要输出变量的内容，需要使用 print 函数
print(qq_number)
print(qq_password)
```

使用解释器执行，如果要输出变量的内容，必须要要使用 print 函数

3.1.3 变量演练 3 —— 超市买苹果

- 可以用 **其他变量的计算结果** 来定义变量
- 变量定义之后，后续就可以直接使用了

需求

- 苹果的价格是 **8.5 元/斤**
- 买了 **7.5 斤** 苹果
- 计算付款金额

```
# 定义苹果价格变量
price = 8.5
# 定义购买重量
weight = 7.5
# 计算金额
money = price * weight
print(money)
```

思考题

- 如果 **只要买苹果，就返 5 块钱**
- 请重新计算购买金额

```
# 定义苹果价格变量
price = 8.5
# 定义购买重量
weight = 7.5
# 计算金额
money = price * weight
# 只要买苹果就返 5 元
money = money - 5
print(money)
```

提问

- 上述代码中，一共定义有几个变量？
  - 三个：price / weight / money

- money = money - 5 是在定义新的变量还是在使用变量？
    - 直接使用之前已经定义的变量
    - 变量名 只有在 **第一次出现** 才是 **定义变量**
    - 变量名 再次出现，不是定义变量，而是直接使用之前定义过的变量
- 在程序开发中，可以修改之前定义变量中保存的值吗？
    - 可以
    - 变量中存储的值，就是可以 **变** 的

3.2. 变量的类型

- 在内存中创建一个变量，会包括：
    1. 变量的名称
    2. 变量保存的数据
    3. 变量存储数据的类型
    4. 变量的地址（标示）

3.2.1 变量类型的演练 —— 个人信息

需求

- 定义变量保存小明的个人信息
- 姓名：**小明**
- 年龄：**18** 岁
- 性别：**是**男生
- 身高：**1.75** 米
- 体重：**75.0** 公斤

利用 **单步调试** 确认变量中保存数据的类型

提问

1. 在演练中，一共有几种数据类型？
    - 4 种
    - str —— 字符串
    - bool —— 布尔（真假）
    - int —— 整数
    - float —— 浮点数（小数）
2. 在 Python 中定义变量时需要指定类型吗？
    - 不需要
    - Python 可以根据 = 等号右侧的值，自动推导出变量中存储数据的类型

3.2.2 变量的类型

在 Python 中定义变量是 **不需要指定类型**（在其他很多高级语言中都需要）

◆ 数据类型根据是否是数字可以分为 **数字型** 和 **非数字型**

- 数字型
  - 整型 (int)
  - 浮点型 (float)
  - 布尔型 (bool)
    - 真 True 非 0 数 —— **非零即真**
    - 假 False 0
  - 复数型 (complex)
    - 主要用于科学计算，例如：平面场问题、波动问题、电感电容等问题
- 非数字型
  - 字符串
  - 列表
  - 元组
  - 字典

提示：在 Python 2.x 中，**整数** 根据保存数值的长度还分为：

- int（整数）
- long（长整数）

- 使用 type 函数可以查看一个变量的类型

```
In [1]: type(name)
```

◆ 数据类型还可以根据是否是组合类型可以分为 **基本类型** 和 **组合类型**

- 基本类型
  - 整型 (int)
  - 浮点型 (float)
  - 布尔型 (bool)
  - 复数型 (complex)
  - 字符串
- 组合类型

- 列表
- 元组
- 集合
- 字典

3.2.3 不同类型变量之间的计算

3.2.3.1 数字型变量 之间可以直接计算

- 在 Python 中，两个数字型变量是可以直接进行 算数运算的
- 如果变量是 bool 型，在计算时
  - True 对应的数字是 1
  - False 对应的数字是 0

演练步骤

1. 定义整数 i = 10
2. 定义浮点数 f = 10.5
3. 定义布尔型 b = True
4. 在 iPython 中，使用上述三个变量相互进行算术运算

3.2.3.2 字符串变量 之间使用 + 拼接字符串

- 在 Python 中，字符串之间可以使用 + 拼接生成新的字符串

```
In [1]: first_name = "三"
In [2]: last_name = "张"
In [3]: first_name + last_name
Out[3]: '三张'
```

3.2.3.3 字符串变量 可以和 整数 使用 * 重复拼接相同的字符串

```
In [1]: "-" * 50
Out[1]: '--------------------------------------------------'
```

3.2.3.4 数字型变量 和 字符串 之间 不能进行其他计算

```
In [1]: first_name = "zhang"
In [2]: x = 10
In [3]: x + first_name
---------------------------------------------------------------------------
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

类型错误：`+` 不支持的操作类型：`int` 和 `str`

3.3 变量的输入

- 所谓 **输入**，就是 **用代码 获取** 用户通过 **键盘** 输入的信息
- 例如：去银行取钱，在 ATM 上输入密码
- 在 Python 中，如果要获取用户在 **键盘** 上的输入信息，需要使用到 `input` 函数

3.3.1 关于函数

- 一个 **提前准备好的功能**(别人或者自己写的代码)，**可以直接使用**，而 **不用关心内部的细节**
- 目前已经学习过的函数

函数	说明
print(x)	将 x 输出到控制台
type(x)	查看 x 的变量类型

3.3.2 input 函数实现键盘输入

- 在 Python 中可以使用 `input` 函数从键盘等待用户的输入
- 用户输入的 **任何内容** Python 都认为是一个 **字符串**
- 语法如下：

```
字符串变量 = input("提示信息：")
```

3.3.3 类型转换函数

函数	说明

函数	说明
int(x)	将 x 转换为一个整数
float(x)	将 x 转换到一个浮点数

3.3.4 变量输入演练 —— 超市买苹果增强版

需求

- **收银员输入** 苹果的价格，单位：**元 / 斤**
- **收银员输入** 用户购买苹果的重量，单位：**斤**
- 计算并且 **输出** 付款金额

演练方式 1

```
# 1. 输入苹果单价
price_str = input("请输入苹果价格：")
# 2. 要求苹果重量
weight_str = input("请输入苹果重量：")
# 3. 计算金额
# 1> 将苹果单价转换成小数
price = float(price_str)
# 2> 将苹果重量转换成小数
weight = float(weight_str)
# 3> 计算付款金额
money = price * weight
print(money)
```

提问

1. 演练中，针对 **价格** 定义了几个变量？
    - **两个**
    - `price_str` 记录用户输入的价格字符串
    - `price` 记录转换后的价格数值
2. **思考** —— 如果开发中，需要用户通过控制台 输入 **很多个 数字**，针对每一个数字都要定义两个变量，**方便吗**？

演练方式 2 —— 买苹果改进版

1. **定义** 一个 **浮点变量** 接收用户输入的同时，就使用 `float` 函数进行转换

```
price = float(input("请输入价格:"))
```

改进后的好处：

- 节约空间，只需要为一个变量分配空间
- 起名字方便，不需要为中间变量起名字

改进后的“缺点”：

- 初学者需要知道，两个函数能够嵌套使用，稍微有一些难度

提示

- 如果输入的不是一个数字，程序执行时会出错，有关数据转换的高级话题，后续会讲！

3.4 eval 函数

演练方式 3 —— 买苹果 eval 版

2. **定义** 一个 **浮点变量** 接收用户输入的同时，就使用 eval 函数进行转换

```
price = eval(input("请输入价格:"))
```

改进后的好处：

- 节约空间，只需要为一个变量分配空间
- 起名字方便，不需要为中间变量起名字

改进后的“缺点”：

- 初学者需要知道，两个函数能够嵌套使用，稍微有一些难度

提示

如果输入的是一个字符串，程序执行时可能会出错，有关数据转换的高级话题，后续会讲！

3.4.1 eval 函数的功能：

- 1. eval 函数的参数必须是字符串。所以经常和 input 联合使用。
- 2. eval 函数的功能，通俗的说，就是先去掉字符串的双引号，使得字符串变成表达式
- 3. eval 函数的最终功能，就是返回表达式经过计算的值

```
la=20
lb = eval（‘la’) #相当于 lb=la
print(lb)
```

```
#结果是 20
```

- 演练 1:

```
la=20
lb = eval(‘la+2’) #相当于 lb=la+2
print(lb)
#结果是 22
```

- 演练 2:

```
lb = eval(‘la+2’) #相当于 lb=la+2, la 赋值，直接报错
print(lb)
#结果是 22
```

3.5 变量的格式化符%输出

苹果单价 9.00 元 / 斤，购买了 5.00 斤，需要支付 45.00 元

- 在 Python 中可以使用 print 函数将信息输出到控制台
- 如果希望输出文字信息的同时，**一起输出 数据**，就需要使用到 **格式化操作符**
- % 被称为 **格式化操作符**，专门用于处理字符串中的格式
  - 包含 % 的字符串，被称为 **格式化字符串**
  - % 和不同的 **字符** 连用，**不同类型的数据** 需要使用 **不同的格式化字符**

表 3-1 格式化字符含义

%s	字符串
%d	有符号十进制整数，%06d 表示输出的整数显示位数，不足的地方使用 0 补全
%f	浮点数，%.2f 表示小数点后只显示两位
%%	输出 %

- 语法格式如下:

```
print("格式化字符串" % 变量 1)
print("格式化字符串" % (变量 1, 变量 2...))
```

3.5.1 格式化输出演练 —— 基本练习

需求

1. 定义字符串变量 name，输出 **我的名字叫 小明，请多多关照！**

2. 定义整数变量 student_no，输出 **我的学号是 000001**
3. 定义小数 price、weight、money，输出 **苹果单价 9.00 元／斤，购买了 5.00 斤，需要支付 45.00 元**
4. 定义一个小数 scale，输出 **数据比例是 10.00%**

```
print("我的名字叫 %s，请多多关照！" % name)
print("我的学号是 %06d" % student_no)
print("苹果单价 %.02f 元／斤，购买 %.02f 斤，需要支付 %.02f 元" % (price, weight, money))
print("数据比例是 %.02f%%" % (scale * 100))
```

第四章 基本数据类型与运算

1. 基本数据类型

- 目标
  整数类型
  浮点类型
  布尔类型
  复数类型
  字符串简介

把 Python 数据类型划分为基本数据类型和组合数据类型，是依据数据内容是否可以分隔出子类型，也就是是否是组合成的来划分的。基本数据类型包括所有数字型和字符串，组合数据类型包括除了字符串类型外的所有非数字型。

1.1 整数类型

Python 中的整数类型可以分为：

- **十进制整数**如，0、-1、9、123
- **十六进制整数**，需要 16 个数字 0、1、2、3、4、5、6、7、8、9、a、b、c、d、e、f 来表示整数，必须以 0x 开头，如 0x10、0xfa、0xabcdef
- **八进制整数**，只需要 8 个数字 0、1、2、3、4、5、6、7 来表示整数，必须以 0o 开头，如 0o35、0o11
- **二进制整数**，只需要 2 个数字 0、1 来表示整数，必须以 0b 开头如，0b101、 0b100

以下那些是正确的整数类型？那些是错误的整数类型？

0xaaa, 0o921, 0b210, 0XAA90, 987, 0b1100, 0o172

整数类型变量，默认打印值是十进制整数，如：

```
>>> n = 0xa1
>>> print(n)
161
>>> m = 0b1010
>>> print(m)
10
```

1.2 浮点类型

浮点数又称小数

- 15.0、0.37、-11.2、1.2e2、314.15e-2

1.3 布尔类型

- 布尔型（bool）
  - 真 True 非 0 数 —— 非零即真
  - 假 False 0

1.4 复数类型

- 复数类型表示数学中的复数。复数有一个基本单位元素 j，叫作“虚数单位”。含有虚数单位的数被称为复数。例如：
- 11.3+4j -5.6+7j 1.23e-4+5.67e+89j
- Python 语言中，复数可以看作是二元有序实数对（a, b），表示为：a + bj，其中，a 是实数部分，简称实部，b 是虚数部分，简称虚部。**虚数部分通过后缀“J”或者“j”来表示**。需要注意，当 b 为 1 时，1 不能省略，即 1j 表示复数，而 j 则表示 Python 程序中的一个变量。
- 复数类型中实部和虚部都是浮点类型，对于复数 z，可以用 z.real 和 z.imag 分别获得它的实数部分和虚数部分

★ 注意：abs()函数，用于复数，是求复数实部和虚部的均方根：

```
>>> abs(3+4j)
5.0
```

1.5 字符串简介

1.5.1 字符串概述

- 用单引号、双引号或三引号界定的符号系列称为字符串
  - 单引号、双引号、三单引号、三双引号可以**互相嵌套**，用来表示复杂字符串 'abc'、'123'、'中国'、"Python"、'''Tom said, "Let's go"'''
  - 字符串属于不可变序列
  - 空字符串表示为''或 ""

- 三引号'''或"""表示的字符串**可以换行**，支持排版较为复杂 的字符串；三引号还可以在程序中表示较长的注释。

1.5.2 字符串合并

```
>>> a = 'abc' + '123'     #生成新字符串
>>> x = '1234"abcd'
>>> x
'1234abcd'
>>> x = x + ',.;'
>>> x
'1234abcd,.;'
>>> x = x'efg'       #不允许这样连接字符串
SyntaxError: invalid syntax
```

1.5.3 常用转义字符

转义字符	含义	转义字符	含义
\b	退格，把光标移动到前一 列位置	\\	一个斜线\
\f	换页符	\’	单引号’
\n	换行符	\”	双引号”
\r	回车	\ooo	3 位八进制数对应的字符
\t	水平制表符	\xhh	2 位十六进制数对应的字符
\v	垂直制表符	\uhhhh	4 位十六进制数表示的 Unicode 字符

- 转义字符用法

```
print('Hello\nWorld') Hello
World
>>> print('\101') A
>>> print('\x41')
A
>>> print("我是\u9648\u798f\u660e")#四位十六进制数表示 Unicode 字符 我是陈福明
```

1.5.3 原始字符串

- 字符串界定符前面加字母 r 或 R 表示**原始字符串**，其中的 特殊字符不进行转义，但字符串的**最后一个字符不能是**。 原始字符串主要用于正则表达式、文件路径或者 URL 的场 合。

```
>>> path = 'C:\Windows\notepad.exe'
>>> print(path)
C:\Windows
otepad.exe
>>> path = r'C:\Windows\notepad.exe' #原始字符串，任何字符都不转义
>>> print(path)
C:\Windows\notepad.exe
```

1.5.4 字符编码

1) Python3 字符编码
Python3 的解释器以“utf-8”作为默认字符串编码，其他编码一律归为字节串。UTF-8 是编码规则，对应的字符集是 Unicode 字符集。
2) 字符串 str 与字节串 bytes 互转：
字节串-->decode('原来的字符编码')-->Unicode 字符串-->encode('新的字符编码')-->字节串。

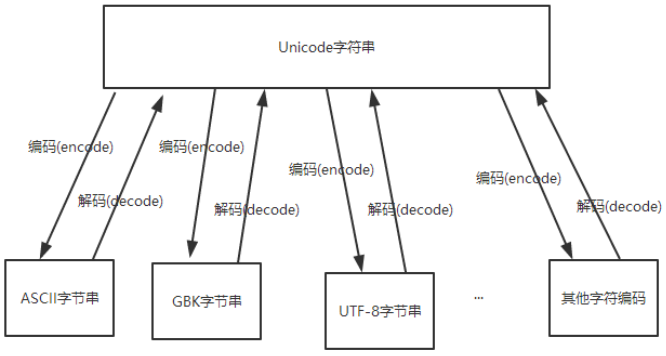

图 5-3 字符串与字节串互转

3) chr(x) 和 ord(x) 函数用于在单字符和 Unicode 编码值之间进行转换

1.6 格式化输出--format()方法

如上一章例题，苹果单价 9.00 元 / 斤，购买了 5.00 斤，需要支付 45.00 元

- format()方法是字符串的一个方法，专门用于处理字符串中的格式
- 字符串 format()方法的语法格式如下:

<模板字符串>.format(<逗号分隔的参数>)

其中,模板字符串是一个由字符串和槽组成的字符串,用来控制字符串和变量的显示效果。槽用大括号({})表示，对应 format()方法中逗号分隔的参数。

```
print("模板字符串".format(变量 1))
print("模板字符串".format (变量 1, 变量 2...))
```

- 如果模板字符串有多个槽，且槽内没有指定序号，则按照槽出现的顺序分别对应.format()方法中的不同参数。
- format()方法中模板字符串的槽除了包括参数序号，还可以包括格式控制信息。
- **{<参数序号>: <格式控制标记>}**
- 其中，格式控制标记用来控制参数显示时的格式 。格式控制标记包括：<填充><对齐><宽度>,<.精度><类型>6 个字段，这些字段都是可选的，可以组合使用

:	<填充>	<对齐>	<宽度>	,	<.精度>	<类型>
引导符号	用于填充的单个字符	< 左对齐 > 右对齐 ^ 居中对齐	槽的设定输出宽度	数字的千位分隔符 适用于整数和浮点数	浮点数小数部分的精度 或 字符串的最大输出长度	整数类型 b, c, d, o, x, X 浮点数类型 e, E, f, %

图 3-2 格式控制标记

- <填充>、<对齐>和<宽度>主要用于对显示格式的规范。
- 宽度指当前槽的设定输出字符宽度，如果该槽参数实际值比宽度设定值大，则使用参数实际长度。如果该值的实际位数小于指定宽度，则按照对齐指定方式在宽度内对齐，默认以空格字符补充。
- 对齐字段分别使用<、>和^三个符号表示左对齐、右对齐和居中对齐。
- 填充字段可以修改默认填充字符，填充字符只能有一个。
- <.精度><类型>主要用于对数值本身的规范
- <.精度>由小数点（.）开头。对于浮点数，精度表示小数部分输出的有效位数。对于字符串，精度表示输出的最大长度。小数点可以理解为对数值的有效截断。
- <类型>表示输出整数和浮点数类型的格式规则。
- 对于整数类型，输出格式包括 6 种：

  1） b：输出整数的二进制方式；

2） c：输出整数对应的 Unicode 字符；
3） d：输出整数的十进制方式；
4） o：输出整数的八进制方式；
5） x：输出整数的小写十六进制方式；
6） X：输出整数的大写十六进制方式；

■ 对于浮点数类型，输出格式包括 4 种：

1） e：输出浮点数对应的小写字母 e 的指数形式；
2） E：输出浮点数对应的大写字母 E 的指数形式；
3） f：输出浮点数的标准浮点形式；
4） %：输出浮点数的百分形式。

1.6.1 输出演练 —— 基本练习

需求

1. 定义字符串变量 name，输出 **我的名字叫 小明，请多多关照！**
2. 定义整数变量 student_no，输出 **我的学号是 000001**
3. 定义小数 price、weight、money，输出 **苹果单价 9.00 元 / 斤，购买了 5.00 斤，需要支付 45.00 元**
4. 定义一个小数 scale，输出 **数据比例是 10.00%**

```
print("我的名字叫{}，请多多关照！".format(name))
print("我的学号是 {:0<6d}".format(student_no))
print("苹果单价 {:0<.2f} 元 / 斤，购买{:0<.2f}斤，需要支付{:0<.2f}元" .format (price, weight, money))
print("数据比例是{:0<.2f}%".format (scale * 100))
```

1.6.2 课后练习 —— 个人名片

需求

- 在控制台依次提示用户输入：**姓名**、**公司**、**职位**、**电话**、**邮箱**
- 按照以下格式输出：

****

公司名称

姓名（职位）

电话：电话

邮箱：邮箱

****

实现代码如下：

```
"""
在控制台依次提示用户输入：姓名、公司、职位、电话、电子邮箱
"""
name = input("请输入姓名：")
company = input("请输入公司：")
title = input("请输入职位：")
phone = input("请输入电话：")
email = input("请输入邮箱：")

print("*" * 50)
print(company)
print()
print("{} ({})" .format(name, title))
print()
print("电话：{}"    .format(phone))
print("邮箱：{}"    .format(email))
print("*" * 50)
```


2. 算数运算符

计算机，顾名思义就是负责进行 **数学计算** 并且 **存储计算结果** 的电子设备

目标

- 算术运算符的基本使用

2.1. 算数运算符

- 算数运算符是 **运算符的一种**

- 是完成基本的算术运算使用的符号，用来处理四则运算

表 4-1 算数运算符

运算符	描述	实例
+	加	10 + 20 = 30
-	减	10 - 20 = -10
*	乘	10 * 20 = 200
/	除	10 / 20 = 0.5
//	取整除	返回除法的整数部分（商） 9 // 2 输出结果 4
%	取余数	返回除法的余数 9 % 2 = 1
**	幂	又称次方、乘方，2 ** 3 = 8

- 在 Python 中 `*` 运算符还可以用于字符串，计算结果就是字符串重复指定次数的结果

```
In [1]: "-" * 50
Out[1]: '--------------------------------------------------'
```

2.2. 算数运算符的优先级

- 和数学中的运算符的优先级一致，在 Python 中进行数学计算时，同样也是：
  - **先乘除后加减**
  - 同级运算符是 **从左至右** 计算
  - 可以使用 `()` 调整计算的优先级
- 以下表格的算数优先级由高到最低顺序排列

表 4-2 算数运算符优先级

运算符	描述

表 4-2 算数运算符优先级

运算符	描述
**	幂（最高优先级）
* / % //	乘、除、取余数、取整除
+ -	加法、减法

- 例如：
  - `2 + 3 * 5 = 17`
  - `(2 + 3) * 5 = 25`
  - `2 * 3 + 5 = 11`
  - `2 * (3 + 5) = 16`

3. 其他运算符

目标

- 比较（关系）运算符
- 逻辑运算符
- 赋值运算符
- 运算符的优先级

数学符号表链接：https://zh.wikipedia.org/wiki/数学符号表

3.1. 比较（关系）运算符

表 5-1 关系运算符

运算符	描述
==	检查两个操作数的值是否 **相等**，如果是，则条件成立，返回 True
!=	检查两个操作数的值是否 **不相等**，如果是，则条件成立，返回 True
>	检查左操作数的值是否 **大于** 右操作数的值，如果是，则条件成立，返回 True
<	检查左操作数的值是否 **小于** 右操作数的值，如果是，则条件成立，返回 True

表 5-1 关系运算符

运算符	描述
>=	检查左操作数的值是否 **大于或等于** 右操作数的值，如果是，则条件成立，返回 True
<=	检查左操作数的值是否 **小于或等于** 右操作数的值，如果是，则条件成立，返回 True

Python 2.x 中判断 **不等于** 还可以使用 <> 运算符

!= 在 Python 2.x 中同样可以用来判断 **不等于**

3.2. 逻辑运算符

表 5-2 逻辑运算符

运算符	逻辑表达式	描述
and	x and y	只有 x 和 y 的值都为 True，才会返回 True 否则只要 x 或者 y 有一个值为 False，就返回 False
or	x or y	只要 x 或者 y 有一个值为 True，就返回 True 只有 x 和 y 的值都为 False，才会返回 False
not	not x	如果 x 为 True，返回 False 如果 x 为 False，返回 True

3.3. 赋值运算符

- 在 Python 中，使用 = 可以给变量赋值
- 在算术运算时，为了简化代码的编写，Python 还提供了一系列的 与 **算术运算符** 对应的 **赋值运算符**
- 注意：**赋值运算符中间不能使用空格**

表 5-3 赋值运算符

运算符	描述	实例
=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为 c
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c *= a 等效于 c = c * a
/=	除法赋值运算符	c /= a 等效于 c = c / a
//=	取整除赋值运算符	c //= a 等效于 c = c // a
%=	取 **模** (余数)赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c *= a *等效于* c = c * a

赋值运算符中有两个重要的知识点：

1) 同步赋值：

```
>>> a, b = 4, 6
>>> print(a, b)
4 6
>>> a, b = b, a
>>> print(a, b)
6 4
```

2） 复合赋值语句与普通赋值语句的比较：

```
>>> x = 3
>>> x *= 3 + 4
>>> x
21
>>> y = 3
>>> y = y * 3 + 4
>>> y
13
```

3.4. 运算符的优先级

- 以下表格的算数优先级由高到最低顺序排列

表 5-4 运算符优先级

运算符	描述
**	幂（最高优先级）
* / % //	乘、除、取余数、取整除
+ -	加法、减法
<= < > >=	比较运算符
== !=	等于运算符
= %= /= //= -= += = *=	赋值运算符
not or and	逻辑运算符

4 常用内置函数

4.1 数学内置函数

表 3-7 数学内置函数

函数	描述	实例	实例结果

表 3-7 数学内置函数

函数	描述	实例	实例结果
abs(a)	求取绝对值	abs(-1)	1
max(list)	求取 list 最大值	max(1,2,3)	3
min(list)	求取 list 最小值	min(1,2,3)	1
sum(list)	求取 list 元素的和	sum(1,2,3)	6
divmod(a,b)	获取商和余数	divmod(5,2)	(2,1)
pow(a,b)	获取乘方数	pow(2,3)	8
round(a,b)	获取指定位数的小数	round(3.1415926,2)	3.14
range(a[,b])	生成一个 a 到 b 的数组，左闭右开	range(1,10)	[1,2,3,4,5,6,7,8,9]
sorted(list)	排序，返回排序后的 list		
len(list)	list 长度	len([1,2,3])	3

4.2 类型转换内置函数

表 3-8 类型转换内置函数

函数	描述	实例	实例结果
int(str)	转换为 int 型	int('168')	168
float(int/str)	将 int 或字符型转换为浮点型	float('2')	2.0
str(int)	转换为字符型	str(134)	'134'
bool(int)	转换为布尔类型	str(0)	False
bytes(str,code)	接收一个字符串，与所要编码的格式，返回一个字节流类型	bytes('abc', 'utf-8') bytes(u'爬虫', 'utf-8')	b'abc' b'\xe7\x88\xac\xe8\x99\xab'
hex(int)	转换为 16 进制	hex(1024)	'0x400'
oct(int)	转换为 8 进制	oct(1024)	'0o2000'
bin(int)	转换为 2 进制	bin(1024)	'0b10000000000'
chr(int)	转换数字为相应 ASCI 码字符	chr(65)	'A'
ord(str)	转换 ASCI 字符为相应的数字	ord('A')	65

5 常用标准库函数

5.1 math 库

内置库 math 库，是常用的数学函数库，包括 acos(), sin()等函数，可以用 dir(math)查看包含那些数学函数和常量:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.ceil(3.6)
4
>>> math.floor(3.6)
3
>>> math.pow(2,3)
8.0
>>> math.sqrt(4)
2.0
>>> math.degrees(3.14)
179.9087476710785
>>> math.radians(180)
3.141592653589793
```

5.2 random 库

内置库 random 库，是常用的随机库，包括产生 0 到 1 之间的小数的 random()函数，列表中随机选择一个元素的 choice()函数，产生随机整数的 randint()函数， 产生正太分布的 uniform()函数，随机取样的 sample()函数，随机乱序的 shuffle()函数等。

```
>>> import random
>>> random.choice(['C++','Java','Python'])
'Java'
>>> random.randint(1,100)
57
>>> random.randrange(0,10,2)
8
>>> random.random()
0.7906454183842933
>>> random.uniform(5,10)
7.753307224388041
>>> random.sample(range(100),10)
[91, 15, 67, 38, 55, 72, 62, 97, 51, 77]
>>> nums=[1001,1002,1003,1004,1005]
>>> random.shuffle(nums)
>>> nums
[1002, 1004, 1005, 1001, 1003]
```

第五章 判断语句

目标

- 开发中的应用场景
- if 语句体验
- if 语句进阶
- 程序的格式框架
- 综合应用

1. 开发中的应用场景

生活中的判断几乎是无所不在的，我们每天都在做各种各样的选择，如果这样？如果那样？……

人生的关键点就是一场场选择……

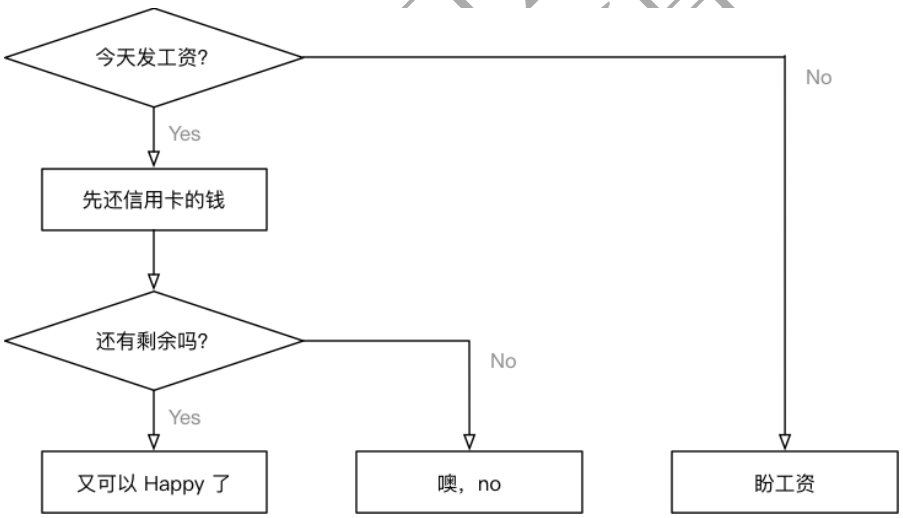

图 4-1 无处不在的选择

1.1 程序中的判断

if 今天发工资:

```
    先还信用卡的钱
    if 有剩余:
        又可以 happy 了，0(∩_∩)0 哈哈~
    else:
        噢，no。。。还的等 30 天
else:
    盼着发工资
```

1.2 判断的定义

- 如果 **条件满足**，才能做某件事情，
- 如果 **条件不满足**，就做另外一件事情，或者什么也不做

正是因为有了判断，才使得程序世界丰富多彩，充满变化！

判断语句 又被称为 “分支语句”，正是因为有了判断，才让程序有了很多的分支

2. if 语句体验

2.1. if 判断语句基本语法

在 Python 中，**if 语句** 就是用来进行判断的，格式如下：

```
if 要判断的条件:
    条件成立时，要做的事情
    ……
```

注意：代码的缩进为一个 tab 键，或者 **4** 个空格 —— **建议使用空格

- 在 Python 开发中，Tab 和空格不要混用！

我们可以把整个 if 语句看成一个完整的代码块

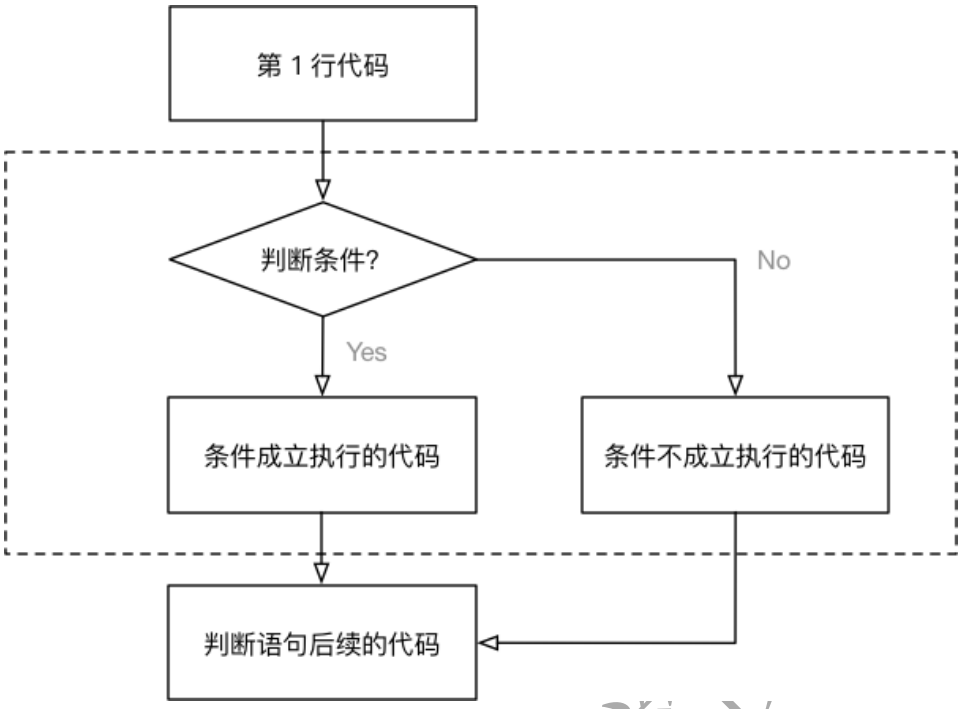

图 4-2 选择结构

2.2 判断语句演练 —— 判断年龄

需求

1. 定义一个整数变量记录年龄
2. 判断是否满 18 岁 （**>=**）
3. 如果满 18 岁，允许进网吧嗨皮

```
# 1. 定义年龄变量
age = 18

# 2. 判断是否满 18 岁
# if 语句以及缩进部分的代码是一个完整的代码块
if age >= 18:
    print("可以进网吧嗨皮……")

# 3. 思考！- 无论条件是否满足都会执行
print("这句代码什么时候执行?")
```

注意：

- if 语句以及缩进部分是一个 **完整的代码块**

2.3 else 处理条件不满足的情况

思考

在使用 if 判断时，只能做到满足条件时要做的事情。那如果需要在 **不满足条件的时候**，做某些事情，该如何做呢？

答案

else，格式如下：

```
if 要判断的条件:
    条件成立时，要做的事情
    ......
else:
    条件不成立时，要做的事情
    ......
```

注意：

- if 和 else 语句以及各自的缩进部分共同是一个 **完整的代码块**

2.4 判断语句演练 —— 判断年龄改进

需求

1. 输入用户年龄
2. 判断是否满 18 岁 （**>=**）
3. 如果满 18 岁，允许进网吧嗨皮
4. 如果未满 18 岁，提示回家写作业

```
# 1. 输入用户年龄
age = int(input("今年多大了？"))

# 2. 判断是否满 18 岁
# if 语句以及缩进部分的代码是一个完整的语法块
if age >= 18:
    print("可以进网吧嗨皮……")
```

```
else:
    print("你还没长大，应该回家写作业！")

# 3. 思考！- 无论条件是否满足都会执行
print("这句代码什么时候执行?")
```

3. 逻辑运算

- 在程序开发中，通常 **在判断条件时**，会需要同时判断多个条件
- 只有多个条件都满足，才能够执行后续代码，这个时候需要使用到 **逻辑运算符**
- **逻辑运算符** 可以把 **多个条件** 按照 **逻辑** 进行 **连接**，变成 **更复杂的条件**
- Python 中的 **逻辑运算符** 包括：**与 and / 或 or / 非 not** 三种

3.1 and

条件 1 and 条件 2

- **与** / 并且**
- 两个条件同时满足，返回 True
- 只要有一个不满足，就返回 False

条件 1条件 2结果
成立 成立 成立
成立 不成立不成立
不成立成立 不成立
不成立不成立不成立

3.2 or

条件 1 or 条件 2

- **或** / 或者**
- 两个条件只要有一个满足，返回 True
- 两个条件都不满足，返回 False

条件 1条件 2结果

条件 1条件 2结果
成立　成立　成立
成立　不成立成立
不成立成立　成立
不成立不成立不成立

3.3 not

not 条件

- **非** / 不是**

条件　结果
成立　不成立
不成立成立

3.3.1 逻辑运算演练

1. 练习 1: 定义一个整数变量 age，编写代码判断年龄是否正确
    - 要求人的年龄在 0-120 之间
2. 练习 2: 定义两个整数变量 python_score、c_score，编写代码判断成绩
    - 要求只要有一门成绩 > 60 分就算合格
3. 练习 3: 定义一个布尔型变量 is_employee，编写代码判断是否是本公司员工
    - 如果不是提示不允许入内

答案 1：

```
# 练习 1: 定义一个整数变量 age，编写代码判断年龄是否正确
age = 100

# 要求人的年龄在 0-120 之间
if age >= 0 and age <= 120:
    print("年龄正确")
else:
    print("年龄不正确")
```

答案 2：

```
# 练习 2: 定义两个整数变量 python_score、c_score，编写代码判断成绩
```

```
python_score = 50
c_score = 50

# 要求只要有一门成绩 > 60 分就算合格
if python_score > 60 or c_score > 60:
    print("考试通过")
else:
    print("再接再厉！")
```

答案 3：

```
# 练习 3: 定义一个布尔型变量 `is_employee`，编写代码判断是否是本公司员工
is_employee = True

# 如果不是提示不允许入内
if not is_employee:
    print("非公勿内")
```

4. if 语句进阶

4.1 elif

- 在开发中，使用 if 可以 **判断条件**
- 使用 else 可以处理 **条件不成立** 的情况
- 但是，如果希望 **再增加一些条件，条件不同，需要执行的代码也不同** 时，就可以使用 elif
- 语法格式如下：

```
if 条件 1:
    条件 1 满足执行的代码
    ……
elif 条件 2:
    条件 2 满足时，执行的代码
    ……
elif 条件 3:
    条件 3 满足时，执行的代码
    ……
else:
    以上条件都不满足时，执行的代码
```

……

- 对比逻辑运算符的代码

```
if 条件1 and 条件2:
    条件1满足 并且 条件2满足 执行的代码
    ……
```

注意

1. elif 和 else 都必须和 if 联合使用，而不能单独使用
2. 可以将 if、elif 和 else 以及各自缩进的代码，看成一个 **完整的代码块**

elif 演练 —— 女友的节日

需求

1. 定义 holiday_name 字符串变量记录节日名称
2. 如果是 **情人节** 应该 **买玫瑰 / 看电影**
3. 如果是 **平安夜** 应该 **买苹果 / 吃大餐**
4. 如果是 **生日** 应该 **买蛋糕**
5. 其他的日子每天都是节日啊……

```
holiday_name = "平安夜"

if holiday_name == "情人节":
    print("买玫瑰")
    print("看电影")
elif holiday_name == "平安夜":
    print("买苹果")
    print("吃大餐")
elif holiday_name == "生日":
    print("买蛋糕")
else:
    print("每天都是节日啊……")
```

4.2 if 的嵌套

elif 的应用场景是：**同时** 判断 **多个条件**，所有的条件是 **平级** 的

- 在开发中，使用 if 进行条件判断，如果希望 **在条件成立的执行语句中** 再 **增加条件判断**，就可以使用 **if 的嵌套**
- **if 的嵌套** 的应用场景就是：**在之前条件满足的前提下，再增加额外的判断**
- **if 的嵌套** 的语法格式，**除了缩进之外** 和之前的没有区别
- 语法格式如下：

```
if 条件 1:
    条件 1 满足执行的代码
    ……

    if 条件 1 基础上的条件 2:
        条件 2 满足时，执行的代码
        ……

    # 条件 2 不满足的处理
    else:
        条件 2 不满足时，执行的代码

# 条件 1 不满足的处理
else:
    条件 1 不满足时，执行的代码
    ……
```

if 的嵌套 演练 —— 火车站安检

需求

1. 定义布尔型变量 has_ticket 表示是否有车票
2. 定义整型变量 knife_length 表示刀的长度，单位：厘米
3. 首先检查是否有车票，如果有，才允许进行 **安检**
4. 安检时，需要检查刀的长度，判断是否超过 20 厘米
    - 如果超过 20 厘米，提示刀的长度，不允许上车
    - 如果不超过 20 厘米，安检通过
5. 如果没有车票，不允许进门

```
# 定义布尔型变量 has_ticket 表示是否有车票
has_ticket = True
# 定义整数型变量 knife_length 表示刀的长度，单位：厘米
knife_length = 20
```

```
# 首先检查是否有车票，如果有，才允许进行 安检
if has_ticket:
    print("有车票，可以开始安检...")
    # 安检时，需要检查刀的长度，判断是否超过 20 厘米
    # 如果超过 20 厘米，提示刀的长度，不允许上车
    if knife_length >= 20:
        print("不允许携带 %d 厘米长的刀上车" % knife_length)
    # 如果不超过 20 厘米，安检通过
    else:
        print("安检通过，祝您旅途愉快……")
# 如果没有车票，不允许进门
else:
print("大哥，您要先买票啊")
```

5 程序的格式框架

冒号和缩进：

- Python 语言采用严格的“缩进”来表明程序的格式框架。缩进指每一行代码开始前的空白区域，用来表示代码之间的**包含和层次关系**。
- 1 个缩进 = 4 个空格
  缩进是 Python 语言中表明程序框架的**唯一手段**

- 当表达分支、循环、函数、类等程序含义时，在 if、while、for、def、class 等保留字所在完整语句后通过英文冒号（:）结尾并在之后进行缩进，表明后续代码与紧邻无缩进语句的所属关系。

```
#e1.1TempConvert.py
TempStr = input("请输入带有符号
if TempStr[-1] in ['F','f']:
    C = (eval(TempStr[0:-1]) -
    print("转换后的温度是{:.2f}C
elif TempStr[-1] in ['C','c']
    F = 1.8*eval(TempStr[0:-1]
    print("转换后的温度是{:.2f}F
else:
    print("输入格式错误")
```

图 4-3 单层缩进

```
DARTS = 1000
hits = 0.0
clock()
for i in range(1, DARTS):
    x, y = random(), random()
    dist = sqrt(x ** 2 + y **
    if dist <= 1.0:
        hits = hits + 1
pi = 4 * (hits/DARTS)
print("Pi的值是{:.2f}".format
```

图 4-4 多层缩进

6. 综合应用 —— 石头剪刀布

目标

1. 强化 **多个条件** 的 **逻辑运算**
2. 体会 `import` 导入模块（“工具包”）的使用

需求

1. 从控制台输入要出的拳 —— 石头（1） / 剪刀（2） / 布（3）
2. 电脑 **随机** 出拳 —— 先假定电脑只会出石头，完成整体代码功能
3. 比较胜负

序号规则

1 石头 胜 剪刀
2 剪刀 胜 布
3 布 胜 石头

6.1 基础代码实现

- 先 **假定电脑就只会出石头**，完成整体代码功能

```
# 从控制台输入要出的拳 —— 石头（1） / 剪刀（2） / 布（3）
player = int(input("请出拳 石头（1） / 剪刀（2） / 布（3）："))
# 电脑 随机 出拳 - 假定电脑永远出石头
computer = 1
# 比较胜负
# 如果条件判断的内容太长，可以在最外侧的条件增加一对大括号
# 再在每一个条件之间，使用回车，PyCharm 可以自动增加 8 个空格
if ((player == 1 and computer == 2) or
        (player == 2 and computer == 3) or
        (player == 3 and computer == 1)):
    print("噢耶！！！电脑弱爆了！！！")
elif player == computer:
    print("心有灵犀，再来一盘！")
else:
    print("不行，我要和你决战到天亮！")
```

6.2 随机数的处理

- 在 Python 中，要使用随机数，首先需要导入 **随机数** 的 **模块** ——“工具包”

```
import random
```

- 导入模块后，可以直接在 **模块名称** 后面敲一个 . 然后按 Tab 键，会提示该模块中包含的所有函数
- random.randint(a, b)，返回 [a, b] 之间的整数，包含 a 和 b
- 例如：

```
random.randint(12, 20)  # 生成的随机数 n: 12 <= n <= 20
random.randint(20, 20)  # 结果永远是 20
random.randint(20, 10)  # 该语句是错误的，下限必须小于上限
```

课后演练：把上例中的 computer = 1 改为 computer = random.randint(1,3)，要记得导入随机模块，即 import random。

第六章 循环语句

目标

- 程序的三大流程
- 循环基本使用
- break 和 continue
- 循环嵌套

1.1. 程序的三大流程

- 在程序开发中，一共有三种流程方式：
  - **顺序** —— **从上向下**，顺序执行代码
  - **分支** —— 根据条件判断，决定执行代码的 **分支**
  - **循环** —— 让 **特定代码** **重复** 执行

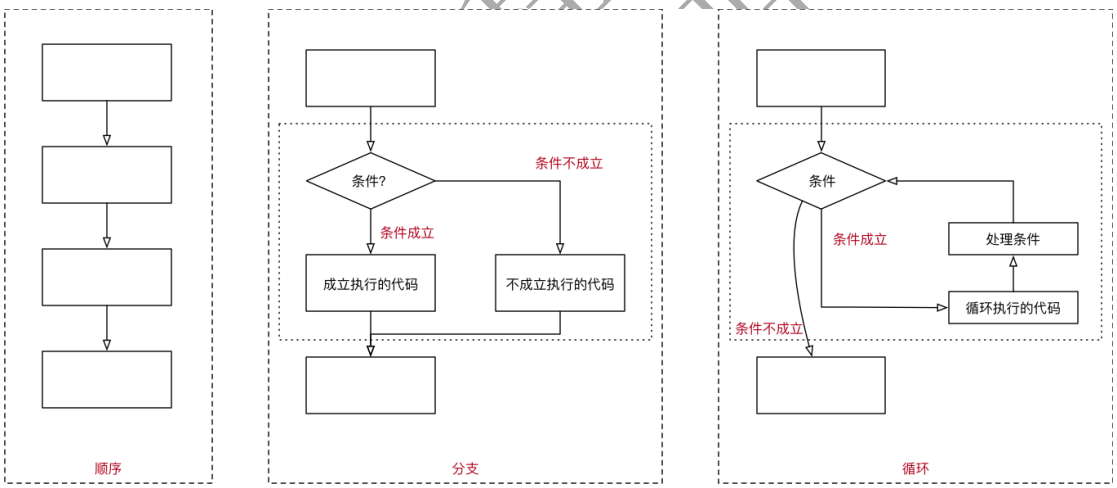

图 5-1 三种流程

1.2. 循环基本使用

- 循环的作用就是让 **指定的代码** 重复的执行
- 循环最常用的应用场景就是 **让执行的代码** 按照 **指定的次数** **重复** 执行
- 需求 —— 打印 5 遍 Hello Python
- 思考 —— 如果要求打印 100 遍怎么办?

1.2.1 while 和 for 语句基本语法

```
初始条件设置 —— 通常是重复执行的 计数器
while 条件(判断 计数器 是否达到 目标次数):
    条件满足时，做的事情 1
    条件满足时，做的事情 2
    条件满足时，做的事情 3
    ...(省略)...
    处理条件(计数器 + 1)
```

注意：

- while 语句以及缩进部分是一个 **完整的代码块**

第一个 循环

需求

- 打印 5 遍 Hello Python

使用 while 循环:

```
#while 循环
# 1. 定义重复次数计数器
i = 1
# 2. 使用 while 判断条件
while i <= 5:
    # 要重复执行的代码
    print("Hello Python")
    # 处理计数器 i
    i = i + 1
print("循环结束后的 i ={} " .format( i))
```

使用 for 循环:

```
#for 循环
#使用 for in 循环
for i in range(5):
    # 要重复执行的代码
```

```
        print("Hello Python")
print("循环结束后的 i ={} " .format( i))
```

注意：循环结束后，之前定义的计数器条件的数值是依旧存在的

死循环

由于程序员的原因，**忘记** 在循环内部 **修改循环的判断条件**，导致循环持续执行，程序无法终止！

1.2.2 赋值运算符

- 在 Python 中，使用 = 可以给变量赋值
- 在算术运算时，为了简化代码的编写，Python 还提供了一系列的 与 **算术运算符** 对应的 **赋值运算符**
- 注意：**赋值运算符中间不能使用空格**

运算符	描述	实例
=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为 c
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c *= a 等效于 c = c * a
/=	除法赋值运算符	c /= a 等效于 c = c / a
//=	取整除赋值运算符	c //= a 等效于 c = c // a
%=	取 **模** (余数)赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c *= a 等效于 c = c * a

1.2.3 Python 中的计数方法

常见的计数方法有两种，可以分别称为：

- **自然计数法**（从 1 开始）—— 更符合人类的习惯
- **程序计数法**（从 0 开始）—— 几乎所有的程序语言都选择从 0 开始计数

因此，大家在编写程序时，应该尽量养成习惯：**除非需求的特殊要求，否则 循环 的计数都从 0 开始**

1.2.4 循环计算

在程序开发中，通常会遇到 **利用循环 重复计算** 的需求

遇到这种需求，可以：

1. 在 `while` 上方定义一个变量，用于 **存放最终计算结果**
2. 在循环体内部，每次循环都用 **最新的计算结果**，**更新** 之前定义的变量

需求

- 计算 0 ~ 100 之间所有数字的累计求和结果

使用 while 循环:

```
# 计算 0 ~ 100 之间所有数字的累计求和结果
# 0. 定义最终结果的变量
result = 0
# 1. 定义一个整数的变量记录循环的次数
i = 0
# 2. 开始循环
while i <= 100:
    print(i)
    # 每一次循环，都让 result 这个变量和 i 这个计数器相加
    result += i
    # 处理计数器
    i += 1
print("0~100 之间的数字求和结果{}".format( result))
```

使用 for 循环:

```
# 计算 0 ~ 100 之间所有数字的累计求和结果
# 0. 定义最终结果的变量
result = 0
#1. 开始循环
for i in range(101):
    print(i)
    # 每一次循环，都让 result 这个变量和 i 这个计数器相加
    result += i
```

```
print("0~100 之间的数字求和结果{}".format( result))
```

需求进阶

- 计算 0 ~ 100 之间 所有 **偶数** 的累计求和结果

开发步骤

1. 编写循环 **确认 要计算的数字**

- 添加 **结果** 变量，在循环内部 **处理计算结果**

使用 while 循环:

```
# 0. 最终结果
result = 0
# 1. 计数器
i = 0
# 2. 开始循环
while i <= 100:
    # 判断偶数
    if i % 2 == 0:
        print(i)
        result += i
    # 处理计数器
    i += 1
print("0~100 之间偶数求和结果 = %d" % result)
```

使用 for 循环:

```
# 0. 最终结果
result = 0
# 1. 开始循环
for i in range(101):
    # 判断偶数
    if i % 2 == 0:
        print(i)
        result += i
```

```
print("0~100 之间偶数求和结果 = %d" % result)
```

1.3. break 和 continue

break 和 continue 是专门在循环中使用的关键字

- break **某一条件满足时**，退出循环，不再执行后续重复的代码
- continue **某一条件满足时**，不执行后续重复的代码

break 和 continue 只针对 **当前所在循环** 有效

1.3.1 break

- **在循环过程中**，如果 **某一个条件满足后**，**不** **再希望** **循环继续执行**，可以使用 break 退出循环

```
i = 0
while i < 10:
    # break 某一条件满足时，退出循环，不再执行后续重复的代码
    # i == 3
    if  i == 3:
        break
    print(i)
    i += 1
print("over")
```

课后演练：上面的代码，改为 for 循环。

break 只针对当前所在循环有效

1.3.2 continue

- **在循环过程中**，如果 **某一个条件满足后**，**不** **希望** **执行循环代码，但是又不希望退出循环**，可以使用 continue

- 也就是：在整个循环中，**只有某些条件**，不需要执行循环代码，而其他条件都需要执行

```
i = 0
while i < 10:
    # 当 i == 7 时，不希望执行需要重复执行的代码
    if i == 7:
        # 在使用 continue 之前，同样应该修改计数器
        # 否则会出现死循环
        i += 1
        continue
    # 重复执行的代码
    print(i)
    i += 1
```

课后演练：上面的代码，改为 for 循环。

- 需要注意：使用 continue 时，**条件处理部分的代码，需要特别注意**，不小心会出现**死循环**

continue 只针对当前所在循环有效

1.4. 循环嵌套

1.4.1 循环嵌套

- while 嵌套就是：while 里面还有 while
- for 嵌套就是：for 里面还有 for
- 也可以混合嵌套

```
while 条件 1:
    条件满足时，做的事情 1
    条件满足时，做的事情 2
    条件满足时，做的事情 3
    ...(省略)...

    while 条件 2:
```

```
        条件满足时，做的事情 1
        条件满足时，做的事情 2
        条件满足时，做的事情 3
        ...(省略)...
        处理条件 2
    处理条件 1
```

1.4.2 循环嵌套演练 —— 九九乘法表

第 1 步：用嵌套打印小星星

需求

- 在控制台连续输出五行 *，每一行星号的数量依次递增

```
*
**
***
****
*****
```

- 使用字符串 * 打印

```
# 1. 定义一个计数器变量，从数字 1 开始，循环会比较方便
row = 1
while row <= 5:
    print("*" * row)
    row += 1
```

课后演练：上面的代码，改为 for 循环。

第 2 步：使用循环嵌套打印小星星

知识点 对 print 函数的使用做一个增强

- 在默认情况下，print 函数输出内容之后，会自动在内容末尾增加换行
- 如果不希望末尾增加换行，可以在 print 函数输出内容的后面增加 ，end=""
- 其中 "" 中间可以指定 print 函数输出内容之后，继续希望显示的内容
- 语法格式如下：

```
# 向控制台输出内容结束之后，不会换行
print("*", end="")
# 单纯的换行
print("")
```

end=""表示向控制台输出内容结束之后，不会换行

假设 Python 没有提供 字符串的 * 操作 拼接字符串

需求

- 在控制台连续输出五行 *，每一行星号的数量依次递增

```
*
**
***
****
*****
```

开发步骤

- 1> 完成 5 行内容的简单输出
- 2> 分析每行内部的 * 应该如何处理？
  - 每行显示的星星和当前所在的行数是一致的
  - 嵌套一个小的循环，专门处理每一行中 列 的星星显示

```
row = 1
while row <= 5:
    # 假设 python 没有提供字符串 * 操作
    # 在循环内部，再增加一个循环，实现每一行的 星星 打印
    col = 1
    while col <= row:
        print("*", end="")
        col += 1
    # 每一行星号输出完成后，再增加一个换行
    print("")
    row += 1
```

课后演练：上面的代码，改为 for 循环。

第 3 步： 九九乘法表

需求 输出 九九乘法表，格式如下：

```
1 * 1 = 1
1 * 2 = 2    2 * 2 = 4
1 * 3 = 3    2 * 3 = 6    3 * 3 = 9
1 * 4 = 4    2 * 4 = 8    3 * 4 = 12    4 * 4 = 16
1 * 5 = 5    2 * 5 = 10    3 * 5 = 15    4 * 5 = 20    5 * 5 = 25
1 * 6 = 6    2 * 6 = 12    3 * 6 = 18    4 * 6 = 24    5 * 6 = 30    6 * 6 = 36
1 * 7 = 7    2 * 7 = 14    3 * 7 = 21    4 * 7 = 28    5 * 7 = 35    6 * 7 = 42    7 * 7 = 49
1 * 8 = 8    2 * 8 = 16    3 * 8 = 24    4 * 8 = 32    5 * 8 = 40    6 * 8 = 48    7 * 8 = 56    8 * 8 = 64
1 * 9 = 9    2 * 9 = 18    3 * 9 = 27    4 * 9 = 36    5 * 9 = 45    6 * 9 = 54    7 * 9 = 63    8 * 9 = 72    9 * 9 = 81
```

开发步骤

- 1. 打印 9 行小星星

```
*
**
***
****
*****
******
*******
********
*********
```

- 2. 将每一个 * 替换成对应的行与列相乘

```
# 定义起始行
row = 1
# 最大打印 9 行
while row <= 9:
    # 定义起始列
    col = 1
    # 最大打印 row 列
```

```
    while col <= row:
        # end = ""，表示输出结束后，不换行
        # "\t"  可以在控制台输出一个制表符，协助在输出文本时对齐
        print("%d * %d = %d" % (col, row, row * col), end="\t")
        #  列数  + 1
        col += 1
    #  一行打印完成的换行
    print("")
    #  行数  + 1
    row += 1
```

课后演练：上面的代码，改为 for 循环。

注意 字符串中的转义字符(复习)

- \t 在控制台输出一个 **制表符**，协助在输出文本时 **垂直方向** 保持对齐
- \n 在控制台输出一个 **换行符**

制表符 的功能是在不使用表格的情况下在 **垂直方向** 按列对齐文本

转义字符	描述
\\	反斜杠符号
\'	单引号
\"	双引号
\n	换行
\t	横向制表符
\r	回车

第七章 程序的异常处理

1. 异常处理

- Python 程序一般对输入有一定要求，但当实际输入不满足程序要求时，可能会产生程序的运行错误。

```
>>>n = eval(input("请输入一个数字: "))
请输入一个整数: python

Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    n = eval(input("请输入一个数字: "))
  File "<string>", line 1, in <module>
NameError: name 'python' is not defined
```

- 由于使用了 eval()函数，如果用户输入不是一个数字则可能报错。这类由于输入与预期不匹配造成的错误有很多种可能，不能逐一列出可能性进行判断。为了保证程序运行的稳定性，这类运行错误应该被程序捕获并合理控制。
- Python 语言使用保留字 try 和 except 进行异常处理，基本的语法格式如下：。

```
try:
    <语句块 1>
except:
    <语句块 2>
```

- 语句块 1 是正常执行的程序内容，当执行这个语句块发生异常时，则执行 except 保留字后面的语句块 2。例如：

```
try:
    n = eval(input("请输入一个数字: "))
    print("输入数字的 3 次方值为: ", n**3)
except:
    print("输入错误，请输入一个数字!")
```

运行结果：

```
>>>
请输入一个数字: 1010
输入数字的 3 次方值为:  1030301000
>>>
```

```
请输入一个数字: python
输入错误，请输入一个数字!
```

- 除了输入之外，异常处理还可以处理程序执行中的运行异常。

```
>>>for i in range(5):
        print(10/i, end=" ")
Traceback (most recent call last):
  File "<pyshell#12>", line 2, in <module>
    print(10/i, end=" ")
ZeroDivisionError: division by zero
```

改为：

```
try:
    for i in range(5):
        print(10/i, end=" ")
except:
    print("某种原因，出错了！ ")
#运行结果：
>>>
某种原因，出错了！
```

2. 异常处理的高级用法

2.1 try/except/else

在 try 语句后也可以跟一个 else 语句，这样当 try 语句块正常执行没有发生异常，则将执行 else 语句后的内容：

```
try:
    pass
except Exception, e:
    print Exception, ":", e
else:
  print( "No exception")
```

2.2 try/except/finally

在 try 语句后边跟一个 finally 语句，则不管 try 语句块有没有发生异常，都会在执行 try 之后执行 finally 语句后的内容：

```
try:
    pass
except Exception,e:
    print( "Exception: ",e)
finally:
    print( "try is done")
```

2.3 raise 抛出异常

使用 raise 来抛出一个异常：

```
a = 0
if a == 0:
    raise Exception("a must not be zero")
```

第八章 高级变量类型

学习目标

- 列表
- 元组
- 字典
- 字符串
- 公共方法

知识点回顾

- Python 中数据类型可以分为 **数字型** 和 **非数字型**
- 数字型
  - 整型 (int)
  - 浮点型 (float)
  - 布尔型 (bool)
    - 真 True 非 0 数 —— **非零即真**
    - 假 False 0
  - 复数型 (complex)
    - 主要用于科学计算，例如：平面场问题、波动问题、电感电容等问题
- 非数字型
  - 字符串
  - 列表
  - 元组
  - 字典
- 在 Python 中，所有 **非数字型变量** 都支持以下特点：

1. 都是一个 **序列** sequence，也可以理解为 **容器**
2. **取值** []
3. **遍历** for in
4. **计算长度、最大/最小值、比较、删除**
5. **链接** + 和 **重复** *
6. **切片**

1. 列表

1.1 列表的定义

- List（列表） 是 Python 中使用 **最频繁** 的数据类型，在其他语言中通常叫做 **数组**
- 专门用于存储 **一串 信息**
- 列表用 [] 定义，**数据** 之间使用 ， 分隔
- 列表的 **索引** 从 0 开始
    - **索引** 就是数据在 **列表** 中的位置编号，**索引** 又可以被称为 **下标**

注意：从列表中取值时，如果 **超出索引范围**，程序会报错

```
name_list = ["zhangsan", "lisi", "wangwu"]
```

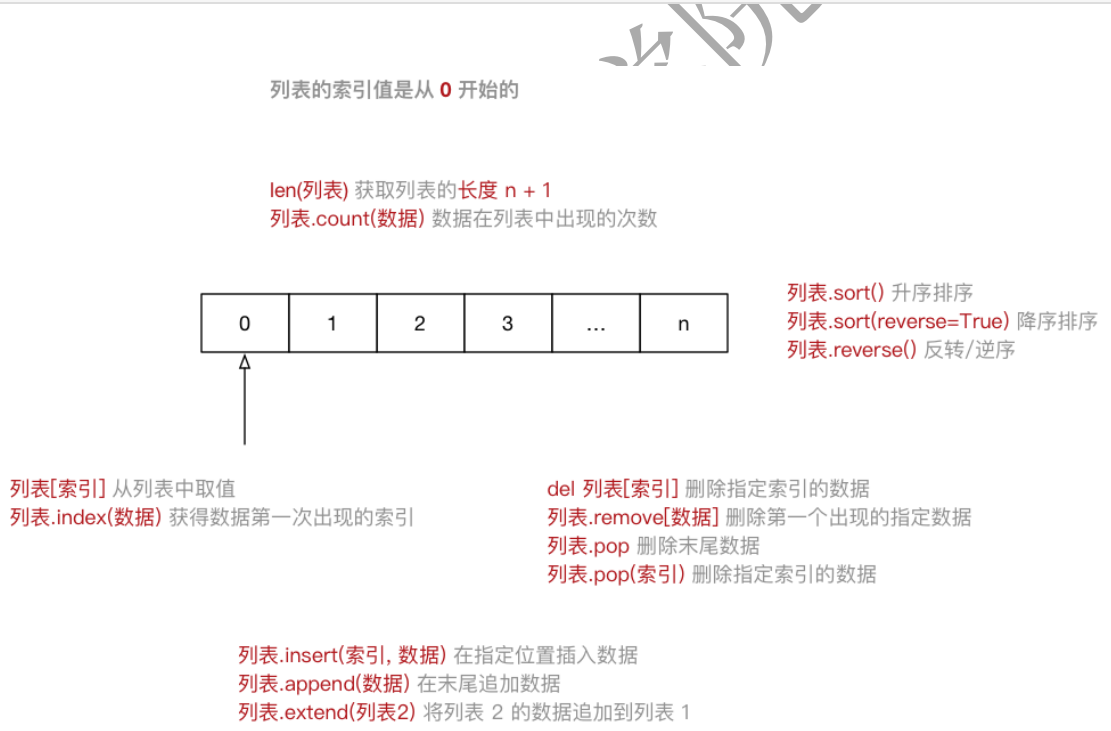

图 5-1 列表

1.2 列表常用操作

- 在 ipython3 中定义一个 **列表**，例如：name_list = []
- 输入 name_list. 按下 TAB 键，ipython 会提示 **列表** 能够使用的 **方法** 如下：

```
In [1]: name_list.
name_list.append     name_list.count      name_list.insert     name_list.reverse
name_list.clear      name_list.extend     name_list.pop        name_list.sort
name_list.copy       name_list.index      name_list.remove
```

序号	分类	关键字 / 函数 / 方法	说明
1	增加	列表.insert(索引，数据)	在指定位置插入数据
|	列表.append(数据)	在末尾追加数据	
|	列表.extend(列表 2)	将列表 2 的数据追加到列表	
2	修改	列表[索引] = 数据	修改指定索引的数据
3	删除	del 列表[索引]	删除指定索引的数据
|	列表.remove[数据]	删除第一个出现的指定数据	
|	列表.pop	删除末尾数据	
|	列表.pop(索引)	删除指定索引数据	
|	列表.clear	清空列表	
4	统计	len(列表)	列表长度
|	列表.count(数据)	数据在列表中出现的次数	
5	排序	列表.sort()	升序排序
|	列表.sort(reverse=True)	降序排序	
|	列表.reverse()	逆序、反转	

del 关键字（科普）

- 使用 del 关键字(delete) 同样可以删除列表中元素
- del 关键字本质上是用来 **将一个变量从内存中删除的**
- 如果使用 del 关键字将变量从内存中删除，后续的代码就不能再使用这个变量了

```
del name_list[1]
```

在日常开发中，要从列表删除数据，建议 **使用列表提供的方法**

关键字、函数和方法（科普）

- **关键字** 是 Python 内置的、具有特殊意义的标识符

```
In [1]: import keyword
In [2]: print(keyword.kwlist)
```

```
In [3]: print(len(keyword.kwlist))
```

关键字后面不需要使用括号

- **函数** 封装了独立功能，可以直接调用

函数名(参数)

函数需要死记硬背

- **方法** 和函数类似，同样是封装了独立的功能
- **方法** 需要通过 **对象** 来调用，表示针对这个 **对象** 要做的操作

对象.方法名(参数)

在变量后面输入 .，然后选择针对这个变量要执行的操作，记忆起来比函数要简单很多

1.3 循环遍历

- **遍历** 就是 **从头到尾 依次** 从 **列表** 中获取数据
  - 在 **循环体内部** 针对 **每一个元素**，执行相同的操作
- 在 Python 中为了提高列表的遍历效率，专门提供的 **迭代 iteration 遍历**
- 使用 for 就能够实现迭代遍历

```
# for 循环内部使用的变量 in 列表
for name in name_list:
    循环内部针对列表元素进行操作
    print(name)
```

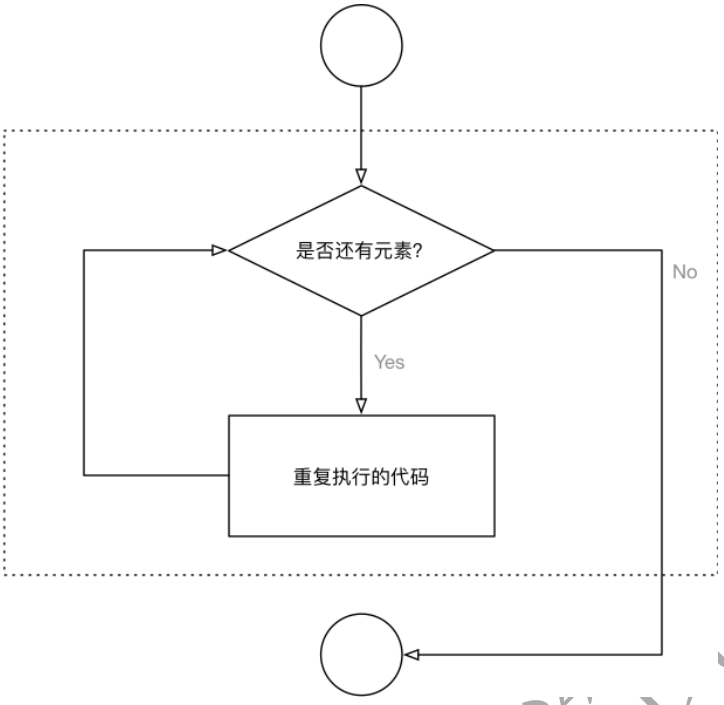

图 5-2 循环遍历

1.4 应用场景

- 尽管 Python 的 **列表** 中可以 **存储不同类型的数据**
- 但是在开发中，更多的应用场景是
    1. **列表** 存储相同类型的数据
    2. 通过 **迭代遍历**，在循环体内部，针对列表中的每一项元素，执行相同的操作

2. 元组

2.1 元组的定义

- Tuple（元组）与列表类似，不同之处在于元组的 **元素不能修改**
    - **元组** 表示多个元素组成的序列
    - **元组** 在 Python 开发中，有特定的应用场景
- 用于存储 **一串 信息**，**数据** 之间使用 ， 分隔
- 元组用 () 定义
- 元组的 **索引** 从 0 开始
    - **索引** 就是数据在 **元组** 中的位置编号

```
info_tuple = ("zhangsan", 18, 1.75)
```

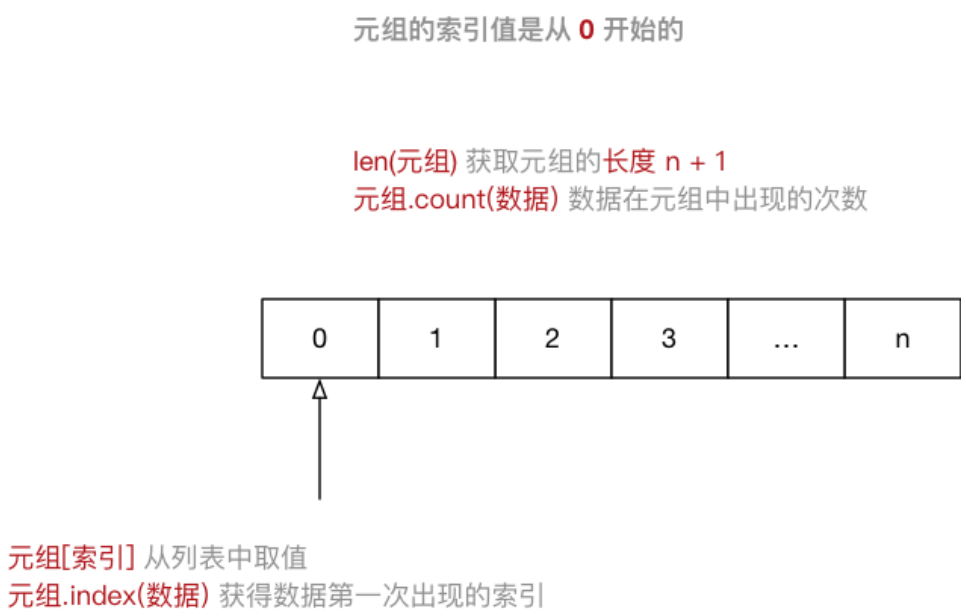

图 5-3 元组

创建空元组

```
info_tuple = ()
```

元组中 只包含一个元素 时，需要 在元素后面添加逗号

```
info_tuple = (50, )
```

2.2 元组常用操作

- 在 ipython3 中定义一个 **元组**，例如：info = ()
- 输入 info. 按下 TAB 键，ipython 会提示 **元组** 能够使用的函数如下：

```
info.count    info.index
```

有关 **元组** 的 **常用操作** 可以参照上图练习

2.3 循环遍历

- **取值** 就是从 **元组** 中获取存储在指定位置的数据
- **遍历** 就是 **从头到尾 依次** 从 **元组** 中获取数据

```
# for 循环内部使用的变量 in 元组
for item in info:

    循环内部针对元组元素进行操作
    print(item)
```

- 在 Python 中，可以使用 for 循环遍历所有非数字型类型的变量：**列表**、**元组**、**字典** 以及 **字符串**
- 提示：在实际开发中，除非 **能够确认元组中的数据类型**，否则针对元组的循环遍历需求并不是很多

2.4 应用场景

- 尽管可以使用 for in 遍历 **元组**
- 但是在开发中，更多的应用场景是：
    - **函数的 参数 和 返回值**，一个函数可以接收 **任意多个参数**，或者 **一次返回多个数据**
        - 有关 **函数的参数 和 返回值**，在后续 **函数高级** 给大家介绍
    - **格式字符串**，格式化字符串后面的 () 本质上就是一个元组
    - **让列表不可以被修改**，以保护数据安全

```
info = ("zhangsan", 18)
print("%s 的年龄是 %d" % info)
```

元组和列表之间的转换

- 使用 list 函数可以把元组转换成列表

list(元组)

- 使用 tuple 函数可以把列表转换成元组

tuple(列表)

3. 字典

3.1 字典的定义

- dictionary（字典） 是 **除列表以外** Python 之中 **最灵活** 的数据类型
- 字典同样可以用来 **存储多个数据**
  - 通常用于存储 **描述一个 物体 的相关信息**
- 和列表的区别
  - **列表** 是 **有序** 的对象集合
  - **字典** 是 **无序** 的对象集合
- 字典用 {} 定义
- 字典使用 **键值对** 存储数据，键值对之间使用 , 分隔
  - **键** key 是索引
  - **值** value 是数据
  - **键** 和 **值** 之间使用 : 分隔
  - **键必须是唯一的**
  - **值 可以取任何数据类型，但 键 只能使用 字符串、数字或 元组**

```
xiaoming = {"name": "小明",
            "age": 18,
            "gender": True,
            "height": 1.75}
```

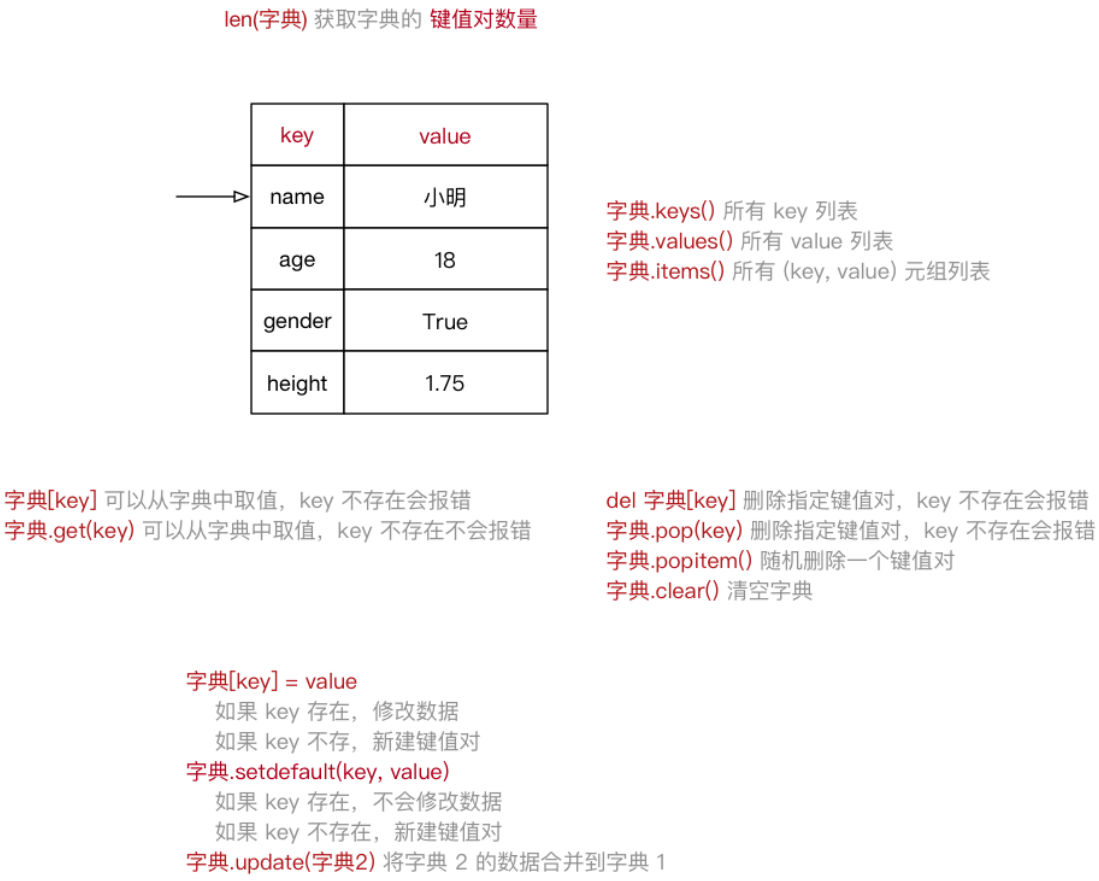

图 5-4 字典

3.2 字典常用操作

- 在 ipython3 中定义一个 **字典**，例如：xiaoming = {}
- 输入 xiaoming. 按下 TAB 键，ipython 会提示 **字典** 能够使用的函数如下：

```
In [1]: xiaoming.
xiaoming.clear       xiaoming.items       xiaoming.setdefault
xiaoming.copy        xiaoming.keys        xiaoming.update
xiaoming.fromkeys    xiaoming.pop         xiaoming.values
xiaoming.get         xiaoming.popitem
```

有关 **字典** 的 **常用操作** 可以参照上图练习

3.3 循环遍历

- **遍历** 就是 **依次** 从 **字典** 中获取所有键值对

```
# for 循环内部使用的 `key 的变量` in 字典
for k in xiaoming:
    print("%s: %s" % (k, xiaoming[k]))
```

提示：在实际开发中，由于字典中每一个键值对保存数据的类型是不同的，所以针对字典的循环遍历需求并不是很多

3.4 应用场景

- 尽管可以使用 for in 遍历 **字典**
- 但是在开发中，更多的应用场景是：
  - 使用 **多个键值对**，存储 **描述一个 物体 的相关信息** —— 描述更复杂的数据信息
  - 将 **多个字典** 放在 **一个列表** 中，再进行遍历，在循环体内部针对每一个字典进行 **相同的处理**

```
card_list = [{"name": "张三",
              "qq": "12345",
              "phone": "110"},
             {"name": "李四",
              "qq": "54321",
              "phone": "10086"}
             ]
```

4. 字符串

4.1 字符串的定义

- **字符串** 就是 **一串字符**，是编程语言中表示文本的数据类型
- 在 Python 中可以使用 **一对双引号** ″ 或者 **一对单引号** ’ 定义一个字符串
  - 虽然可以使用 \″ 或者 \’ 做字符串的转义，但是在实际开发中：
    - 如果字符串内部需要使用 ″，可以使用 ’ 定义字符串

    - 如果字符串内部需要使用 ’，可以使用 〞定义字符串
- 可以使用 **索引** 获取一个字符串中 **指定位置的字符**，索引计数从 **0** 开始
- 也可以使用 for **循环遍历** 字符串中每一个字符

大多数编程语言都是用 〞来定义字符串

```
string = "Hello Python"
for c in string:
print(c)
```

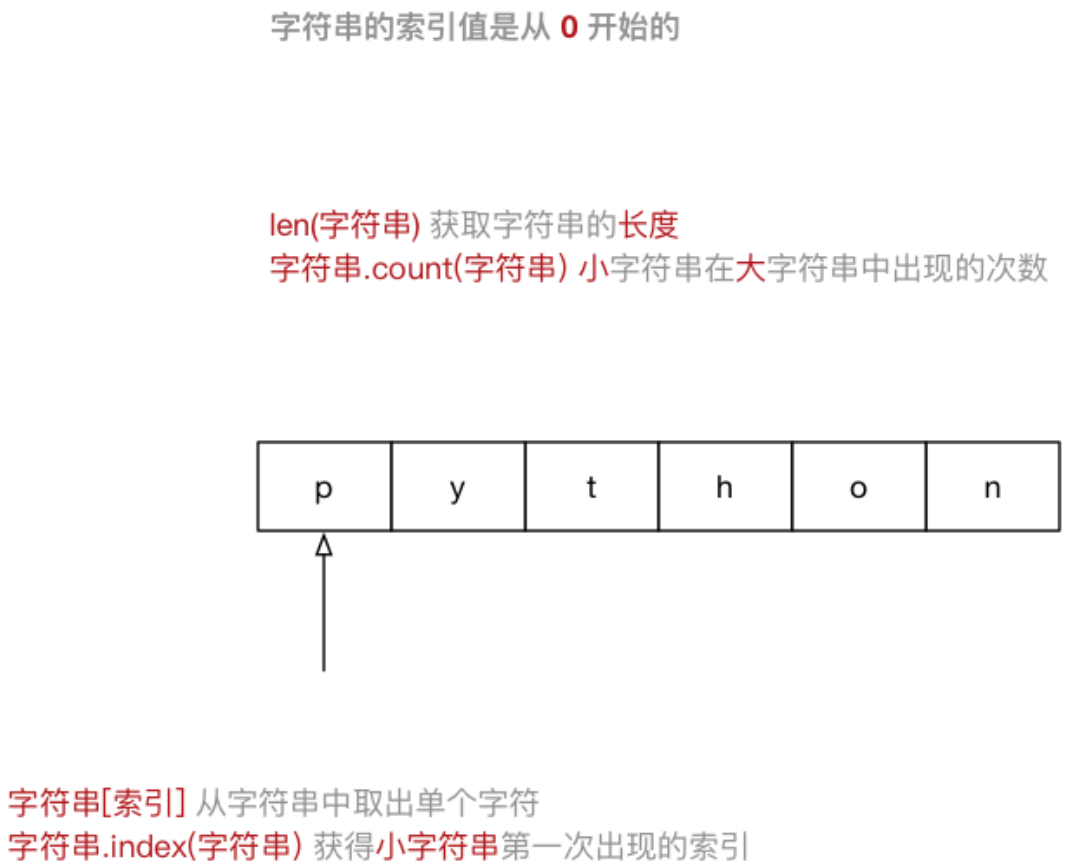

图 5-5 字符串

4.2 字符串的常用操作

- 在 ipython3 中定义一个 **字符串**，例如：hello_str = 〞〞

- 输入 hello_str. 按下 TAB 键，ipython 会提示 **字符串** 能够使用的 **方法** 如下：

```
In [1]: hello_str.
hello_str.capitalize      hello_str.isidentifier   hello_str.rindex
hello_str.casefold        hello_str.islower        hello_str.rjust
hello_str.center          hello_str.isnumeric      hello_str.rpartition
hello_str.count           hello_str.isprintable    hello_str.rsplit
hello_str.encode          hello_str.isspace        hello_str.rstrip
hello_str.endswith        hello_str.istitle        hello_str.split
hello_str.expandtabs      hello_str.isupper        hello_str.splitlines
hello_str.find            hello_str.join           hello_str.startswith
hello_str.format          hello_str.ljust          hello_str.strip
hello_str.format_map      hello_str.lower          hello_str.swapcase
hello_str.index           hello_str.lstrip         hello_str.title
hello_str.isalnum         hello_str.maketrans      hello_str.translate
hello_str.isalpha         hello_str.partition      hello_str.upper
hello_str.isdecimal       hello_str.replace        hello_str.zfill
hello_str.isdigit         hello_str.rfind
```

提示：正是因为 python 内置提供的方法足够多，才使得在开发时，能够针对字符串进行更加灵活的操作！应对更多的开发需求！

1) 判断类型 - 9

方法	说明
string.isspace()	如果 string 中只包含空格，则返回 True
string.isalnum()	如果 string 至少有一个字符并且所有字符都是字母或数字则返回 True
string.isalpha()	如果 string 至少有一个字符并且所有字符都是字母则返回 True
string.isdecimal()	如果 string 只包含数字则返回 True，全角数字
string.isdigit()	如果 string 只包含数字则返回 True，全角数字、⑴、\u00b2
string.isnumeric()	如果 string 只包含数字则返回 True，全角数字，汉字数字
string.istitle()	如果 string 是标题化的(每个单词的首字母大写)则返回 True
string.islower()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True
string.isupper()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True

```
>>> "123 一二三".isnumeric()
True
```

```
>>> "123 一二三".isdecimal()
False
>>> "123 1  2  3 ".isdecimal()
True
```

2) 查找和替换 - 7

方法	说明
string.startswith(str)	检查字符串是否是以 str 开头，是则返回 True
string.endswith(str)	检查字符串是否是以 str 结束，是则返回 True
string.find(str, start=0, end=len(string))	检测 str 是否包含在 string 中，如果 start 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回 -1
string.rfind(str, start=0, end=len(string))	类似于 find()，不过是从右边开始查找
string.index(str, start=0, end=len(string))	跟 find() 方法类似，不过如果 str 不在 string 会报错
string.rindex(str, start=0, end=len(string))	类似于 index()，不过是从右边开始
string.replace(old_str, new_str, num=string.count(old))	把 string 中的 old_str 替换成 new_str，如果 num 指定，则替换不超过 num 次

3) 大小写转换 - 5

方法	说明
string.capitalize()	把字符串的第一个字符大写
string.title()	把字符串的每个单词首字母大写
string.lower()	转换 string 中所有大写字符为小写
string.upper()	转换 string 中的小写字母为大写
string.swapcase()	翻转 string 中的大小写

4) 文本对齐 - 3

方法	说明
string.ljust(width)	返回一个原字符串左对齐，并使用空格填充至长度 width 的新字符串
string.rjust(width)	返回一个原字符串右对齐，并使用空格填充至长度 width 的新字符串
string.center(width)	返回一个原字符串居中，并使用空格填充至长度 width 的新字符串

5) 去除空白字符 - 3

方法	说明
string.lstrip()	截掉 string 左边（开始）的空白字符
string.rstrip()	截掉 string 右边（末尾）的空白字符
string.strip()	截掉 string 左右两边的空白字符

6) 拆分和连接 - 5

方法	说明
string.partition(str)	把字符串 string 分成一个 3 元素的元组 (str 前面, str, str 后面)
string.rpartition(str)	类似于 partition() 方法，不过是从右边开始查找
string.split(str="", num)	以 str 为分隔符拆分 string，如果 num 有指定值，则仅分隔 num + 1 个子字符串，str 默认包含 '\r', '\t', '\n' 和空格
string.splitlines()	按照行('\r', '\n', '\r\n')分隔，返回一个包含各行作为元素的列表
string.join(seq)	以 string 作为分隔符，将 seq 中所有的元素（的字符串表示）合并为一个新的字符串

4.3 字符串的切片

- **切片** 方法适用于 **字符串**、**列表**、**元组**
  - **切片** 使用 **索引值** 来限定范围，从一个大的 **字符串** 中 **切出** 小的 **字符串**
  - **列表** 和 **元组** 都是 **有序** 的集合，都能够 **通过索引值** 获取到对应的数据
  - **字典** 是一个 **无序** 的集合，是使用 **键值对** 保存数据

字符串[开始索引:结束索引:步长]

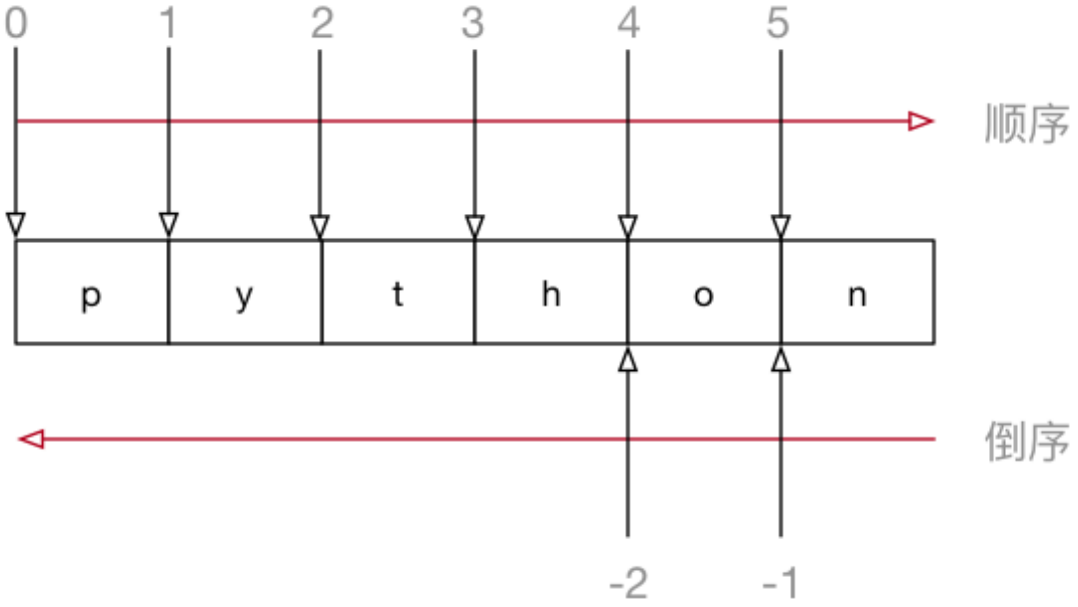

图 5-6 字符串切片

注意：

1. 指定的区间属于 **左闭右开** 型 [开始索引，结束索引) => 开始索引 >= 范围 < 结束索引
    - 从 起始 位开始，到 **结束位的前一位** 结束（**不包含结束位本身**）
2. 从头开始，**开始索引 数字可以省略，冒号不能省略**
3. 到末尾结束，**结束索引 数字可以省略，冒号不能省略**
4. 步长默认为 1，如果连续切片，**数字和冒号都可以省略**

索引的顺序和倒序

- 在 Python 中不仅支持 **顺序索引**，同时还支持 **倒序索引**
- 所谓倒序索引就是 **从右向左** 计算索引
    - 最右边的索引值是 **-1**，依次递减

演练需求

1. 截取从 2 ~ 5 位置 的字符串
2. 截取从 2 ~ 末尾 的字符串
3. 截取从 开始 ~ 5 位置 的字符串

4. 截取完整的字符串

5. 从开始位置，每隔一个字符截取字符串

6. 从索引 1 开始，每隔一个取一个

7. 截取从 2 ~ 末尾 - 1 的字符串

8. 截取字符串末尾两个字符

9. 字符串的逆序（面试题）

答案

```
num_str = "0123456789"
# 1. 截取从 2 ~ 5 位置 的字符串
print(num_str[2:6])
# 2. 截取从 2 ~ `末尾` 的字符串
print(num_str[2:])
# 3. 截取从 `开始` ~ 5 位置 的字符串
print(num_str[:6])
# 4. 截取完整的字符串
print(num_str[:])
# 5. 从开始位置，每隔一个字符截取字符串
print(num_str[::2])
# 6. 从索引 1 开始，每隔一个取一个
print(num_str[1::2])
# 倒序切片
# -1 表示倒数第一个字符
print(num_str[-1])
# 7. 截取从 2 ~ `末尾 - 1` 的字符串
print(num_str[2:-1])
# 8. 截取字符串末尾两个字符
print(num_str[-2:])
# 9. 字符串的逆序（面试题）
print(num_str[::-1])
```

5. 公共方法

5.1 Python 内置函数

Python 包含了以下内置函数：

函数	描述	备注
len(item)	计算容器中元素个数	
del(item)	删除变量	del 有两种方式
max(item)	返回容器中元素最大值	如果是字典，只针对 key 比较
min(item)	返回容器中元素最小值	如果是字典，只针对 key 比较
cmp(item1, item2)	比较两个值，-1 小于/0 相等/1 大于	Python 3.x 取消了 cmp 函数

注意

- **字符串** 比较符合以下规则： "0" < "A" < "a"

5.2 切片

描述	Python 表达式	结果	支持的数据类型
切片	"0123456789"[::-2]	"97531"	字符串、列表、元组

- **切片** 使用 **索引值** 来限定范围，从一个大的 **字符串** 中 **切出** 小的 **字符串**
- **列表** 和 **元组** 都是 **有序** 的集合，都能够 **通过索引值** 获取到对应的数据
- **字典** 是一个 **无序** 的集合，是使用 **键值对** 保存数据

5.3 运算符

运算符	Python 表达式	结果	描述	支持的数据类型
+	[1, 2] + [3, 4]	[1, 2, 3, 4]	合并	字符串、列表、元组
*	["Hi!"] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	重复	字符串、列表、元组
in	3 in (1, 2, 3)	True	元素是否存在	字符串、列表、元组、字典
not in	4 not in (1, 2, 3)	True	元素是否不存在	字符串、列表、元组、字典
> >= == < <=	(1, 2, 3) < (2, 2, 3)	True	元素比较	字符串、列表、元组

注意

- `in` 在对 **字典** 操作时，判断的是 **字典的键**
- `in` 和 `not in` 被称为 **成员运算符**

成员运算符

成员运算符用于 **测试** 序列中是否包含指定的 **成员**

运算符	描述	实例
in	如果在指定的序列中找到值返回 True，否则返回 False	3 in (1, 2, 3) 返回 True
not in	如果在指定的序列中没有找到值返回 True，否则返回 False	3 not in (1, 2, 3) 返回 False

注意：在对 **字典** 操作时，判断的是 **字典的键**

5.4 完整的 for 循环语法

- 在 Python 中完整的 for 循环 的语法如下：

```
for 变量 in 集合:
    循环体代码
else:
    没有通过 break 退出循环，循环结束后，会执行的代码
```

应用场景

- 在 **迭代遍历** 嵌套的数据类型时，例如 **一个列表包含了多个字典**
- 需求：要判断 某一个字典中 是否存在 指定的 值
    - 如果 **存在**，提示并且退出循环
    - 如果 **不存在**，在 **循环整体结束** 后，希望 **得到一个统一的提示**

```
students = [
    {"name": "阿土",
     "age": 20,
     "gender": True,
     "height": 1.7,
     "weight": 75.0},
    {"name": "小美",
     "age": 19,
     "gender": False,
     "height": 1.6,
     "weight": 45.0},
]
```

```
find_name = "阿土"
for stu_dict in students:
    print(stu_dict)
    # 判断当前遍历的字典中姓名是否为 find_name
    if stu_dict["name"] == find_name:
        print("找到了")
        # 如果已经找到，直接退出循环，就不需要再对后续的数据进行比较
        break
else:
    print("没有找到")
print("循环结束")
```

第九章 函数

目标

- 函数的快速体验
- 函数的基本使用
- 函数的参数
- 函数的返回值
- 函数的嵌套调用
- 在模块中定义函数
- lamda 函数
- 变量作用域

1. 函数的快速体验

1.1 快速体验

- 所谓**函数**，就是把 **具有独立功能的代码块** 组织为一个小模块，在需要的时候 **调用**
- 函数的使用包含两个步骤：
  1. 定义函数 —— **封装** 独立的功能
  2. 调用函数 —— 享受 **封装** 的成果
- **函数的作用**，在开发程序时，使用函数可以提高编写的效率以及代码的 **重用**

演练步骤

1. 新建 `04_函数` 项目
2. 复制之前完成的 **乘法表** 文件
3. 修改文件，增加函数定义 `multiple_table()`：
4. 新建另外一个文件，使用 `import` 导入并且调用函数

2. 函数基本使用

2.1 函数的定义

定义函数的格式如下：

```
def 函数名():
    函数封装的代码
    .....
```

1. def 是英文 define 的缩写
2. **函数名称** 应该能够表达 **函数封装代码** 的功能，方便后续的调用
3. **函数名称** 的命名应该 **符合 标识符的命名规则**
    - 可以由 **字母**、**下划线** 和 **数字** 组成
    - **不能以数字开头**
    - **不能与关键字重名**

2.2 函数调用

调用函数很简单的，通过 函数名() 即可完成对函数的调用

2.3 第一个函数演练

需求

- 编写一个打招呼 say_hello 的函数，封装三行打招呼的代码，在函数下方调用打招呼的代码

```
name = "小明"
# 解释器知道这里定义了一个函数
def say_hello():
    print("hello 1")
    print("hello 2")
    print("hello 3")
print(name)
# 只有在调用函数时，之前定义的函数才会被执行
# 函数执行完成之后，会重新回到之前的程序中，继续执行后续的代码
say_hello()
print(name)
```

用 **单步执行 F8 和 F7** 观察以下代码的执行过程

- 定义好函数之后，只表示这个函数封装了一段代码而已
- 如果不主动调用函数，函数是不会主动执行的

思考

- 能否将 **函数调用** 放在 **函数定义** 的上方？
  - 不能！
  - 因为在 **使用函数名** 调用函数之前，必须要保证 Python 已经知道函数的存在
  - 否则控制台会提示 NameError: name 'say_hello' is not defined **(名称错误：say_hello 这个名字没有被定义)**

2.4 PyCharm 的调试工具

- **F8 Step Over** 可以单步执行代码，会把函数调用看作是一行代码直接执行
- **F7 Step Into** 可以单步执行代码，如果是函数，会进入函数内部

2.5 函数的文档注释

- 在开发中，如果希望给函数添加注释，应该在 **定义函数** 的下方，使用 **连续的三对引号**
- 在 **连续的三对引号** 之间编写对函数的说明文字
- 在 **函数调用** 位置，使用快捷键 CTRL + Q 可以查看函数的说明信息

注意：因为 **函数体相对比较独立，函数定义的上方，**应该和其他代码（包括注释）保留 **两个空行**

3. 函数的参数

演练需求

1. 开发一个 sum_2_num 的函数
2. 函数能够实现 **两个数字的求和** 功能

演练代码如下：

```
def sum_2_num():

    num1 = 10
    num2 = 20
    result = num1 + num2
```

```
    print("%d + %d = %d" % (num1, num2, result))

sum_2_num()
```

思考一下存在什么问题

函数只能处理 **固定数值** 的相加

如何解决？

- 如果能够把需要计算的数字，在调用函数时，传递到函数内部就好了！

3.1 函数参数的使用

- 在函数名的后面的小括号内部填写 **参数**
- 多个参数之间使用 ， 分隔

```
def sum_2_num(num1, num2):

    result = num1 + num2

    print("%d + %d = %d" % (num1, num2, result))

sum_2_num(50, 20)
```

3.2 参数的作用

- **函数**，把 **具有独立功能的代码块** 组织为一个小模块，在需要的时候 **调用**
- **函数的参数**，增加函数的 **通用性**，针对 **相同的数据处理逻辑**，能够 **适应更多的数据**
    1. 在函数 **内部**，把参数当做 **变量** 使用，进行需要的数据处理
    2. 函数调用时，按照函数定义的**参数顺序**，把 **希望在函数内部处理的数据**，**通过参数** 传递

3.3 形参和实参

- **形参**：**定义** 函数时，小括号中的参数，是用来接收参数用的，在函数内部 **作为变量使用**
- **实参**：**调用** 函数时，小括号中的参数，是用来把数据传递到 **函数内部** 用的

4. 函数的返回值

- 在程序开发中，有时候，会希望 **一个函数执行结束后，告诉调用者一个结果**，以便调用者针对具体的结果做后续的处理
- **返回值** 是函数 **完成工作**后，**最后** 给调用者的 **一个结果**
- 在函数中使用 return 关键字可以返回结果
- 调用函数一方，可以 **使用变量** 来 **接收** 函数的返回结果

注意：return 表示返回，后续的代码都不会被执行

```
def sum_2_num(num1, num2):
    """对两个数字的求和"""
    return num1 + num2
# 调用函数，并使用 result 变量接收计算结果
result = sum_2_num(10, 20)
print("计算结果是 %d" % result)
```

5. 函数的嵌套调用

- 一个函数里面 **又调用** 了 **另外一个函数**，这就是 **函数嵌套调用**
- 如果函数 test2 中，调用了另外一个函数 test1
    - 那么执行到调用 test1 函数时，会先把函数 test1 中的任务都执行完
    - 才会回到 test2 中调用函数 test1 的位置，继续执行后续的代码

```
def test1():
    print("*" * 50)
    print("test 1")
    print("*" * 50)
def test2():
    print("-" * 50)
    print("test 2")
    test1()
```

```
    print("-" * 50)
test2()
```

函数嵌套的演练 —— 打印分隔线

体会一下工作中 **需求是多变** 的

需求 1

- 定义一个 `print_line` 函数能够打印 `*` 组成的 **一条分隔线**

```
def print_line(char):
    print("*" * 50)
```

需求 2

- 定义一个函数能够打印 **由任意字符组成** 的分隔线

```
def print_line(char):
    print(char * 50)
```

需求 3

- 定义一个函数能够打印 **任意重复次数** 的分隔线

```
def print_line(char, times):
    print(char * times)
```

需求 4

- 定义一个函数能够打印 **5 行** 的分隔线，分隔线要求符合**需求 3**

提示：工作中针对需求的变化，应该冷静思考，**不要轻易修改之前已经完成的，能够正常执行的函数**！

```
def print_line(char, times):
    print(char * times)
def print_lines(char, times):
```

```
row = 0
while row < 5:
    print_line(char, times)
    row += 1
```

6. 使用模块中的函数

模块是 Python 程序架构的一个核心概念

- **模块** 就好比是 **工具包**，要想使用这个工具包中的工具，就需要 **导入 import** 这个模块
- 每一个以扩展名 py 结尾的 Python 源代码文件都是一个 **模块**
- 在模块中定义的 **全局变量** 、 **函数** 都是模块能够提供给外界直接使用的工具

6.1 第一个模块体验

步骤

- 新建分隔线模块 hm_10_module1.py
  - 复制 hm_09_打印多条分隔线.py 中的内容，**最后一行 print 代码除外**
  - 增加一个字符串变量

```
name = "黑马程序员"
```

- 新建体验模块 hm_10_ module2.py 文件，并且编写以下代码：

```
import hm_10_ module1

hm_10_ module1.print_line("-", 80)
print(hm_10_ module1.name)
```

体验小结

- 可以 **在一个 Python 文件 中 定义 变量 或者 函数**
- 然后在 **另外一个文件中** 使用 import 导入这个模块
- 导入之后，就可以使用 模块名.变量 / 模块名.函数 的方式，使用这个模块中定义的变量或者函数

模块可以让 **曾经编写过的代码** 方便的被 **复用**！

6.2 模块名也是一个标识符

- 标示符可以由 **字母**、**下划线** 和 **数字** 组成
- **不能以数字开头**
- **不能与关键字重名**

注意：如果在给 Python 文件起名时，**以数字开头** 是无法在 PyCharm 中通过导入这个模块的

6.3 Pyc 文件（了解）

C 是 compiled **编译过** 的意思

操作步骤

1. 浏览程序目录会发现一个 __pycache__ 的目录
2. 目录下会有一个 hm_10_分隔线模块.cpython-35.pyc 文件，cpython-35 表示 Python 解释器的版本
3. 这个 pyc 文件是由 Python 解释器将 **模块的源码** 转换为 **字节码**
    - Python 这样保存 **字节码** 是作为一种启动 **速度的优化**

字节码

- Python 在解释源程序时是分成两个步骤的
    1. 首先处理源代码，**编译** 生成一个二进制 **字节码**
    2. 再对 **字节码** 进行处理，才会生成 CPU 能够识别的 **机器码**
- 有了模块的字节码文件之后，下一次运行程序时，如果在 **上次保存字节码之后** 没有修改过源代码，Python 将会加载 .pyc 文件并跳过编译这个步骤
- 当 Python 重编译时，它会自动检查源文件和字节码文件的时间戳
- 如果你又修改了源代码，下次程序运行时，字节码将自动重新创建

提示：有关模块以及模块的其他导入方式，后续课程还会逐渐展开！

模块是 Python 程序架构的一个核心概念

7. lamda 函数

```
def func(x,y,callback):
z= x*y+callback(x+y)
return z
#调用
func（2,3,lambda x:x**2）
即：x*y+(x+y)**2 的结果
lambda x:x**2 等价于：
def ???(x):
return x**2
```

8. 变量作用域

- 根据程序中变量所在的位置和作用范围，变量分为**局部变量**和**全局变量**。

8.1 局部变量

- 局部变量仅在函数内部，且作用域也在函数内部，全局变量的作用域跨越多个函数。
- 局部变量指在函数内部使用的变量，仅在函数内部有效，当函数退出时变量将不再存在。

```
>>>def multiply(x, y = 10):
        z = x*y      # z 是函数内部的局部变量
        return z
>>>s = multiply(99, 2)
>>>print(s)
198
>>>print(z)
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print(z)
NameError: name 'z' is not defined
```

- 变量 z 是函数 multiple()内部使用的变量，当函数调用后，变量 z 将不存在。

8.2 全局变量

- 全局变量指在函数之外定义的变量，在程序执行全过程有效。全部变量在函数内部使用

时，需要提前使用保留字 global 声明，语法形式如下：

global <全局变量>

```
>>>n = 2       #n 是全局变量
>>>def multiply(x, y = 10):
        global n
        return x*y*n        #  使用全局变量 n
>>>s = multiply(99, 2)
>>>print(s)
396
```

- 上例中，变量 n 是全局变量，在函数 multiply()中使用时需要在函数内部使用 global 声明，定义后即可使用。
- 如果未使用保留字 global 声明，即使名称相同，也不是全局变量。

```
>>>n = 2       #n 是全局变量
>>>def multiply(x, y = 10):
        n = x*y
        return n        #  此处的 n 不是全局变量
>>>s = multiply(99, 2)
>>>print(s)
198
>>>print(n)     #不改变外部全局变量的值
2
```

第十章 综合应用 —— 名片管理系统

目标

综合应用已经学习过的知识点：

- **变量**
- **流程控制**
- **函数**
- **模块**

开发 **名片管理系统**

系统需求

1. 程序启动，显示名片管理系统欢迎界面，并显示功能菜单

```
**********************************************
欢迎使用【名片管理系统】V1.0

1．新建名片
2．显示全部
3．查询名片

0．退出系统
**********************************************
```

2. 用户用数字选择不同的功能
3. 根据功能选择，执行不同的功能
4. 用户名片需要记录用户的 **姓名、电话、QQ、邮件**
5. 如果查询到指定的名片，用户可以选择 **修改** 或者 **删除** 名片

步骤

1. 框架搭建
2. 新增名片

3. 显示所有名片
4. 查询名片
5. 查询成功后修改、删除名片
6. 让 Python 程序能够直接运行

1. 框架搭建

目标

- 搭建名片管理系统 **框架结构**
    1. **准备文件**，确定文件名，保证能够 **在需要的位置** 编写代码
    2. 编写 **主运行循环**，实现基本的 **用户输入和判断**

1.1 文件准备

1. 新建 `cards_main.py` 保存 **主程序功能代码**
    - 程序的入口
    - 每一次启动名片管理系统都通过 `main` 这个文件启动
2. 新建 `cards_tools.py` 保存 **所有名片功能函数**
    - 将对名片的 **新增、查询、修改、删除** 等功能封装在不同的函数中

1.2 编写主运行循环

- 在 `cards_main` 中添加一个 **无限循环**

```
while True:

    # TODO(小明) 显示系统菜单

    action = input("请选择操作功能：")

    print("您选择的操作是：%s" % action)

    # 根据用户输入决定后续的操作
    if action in ["1", "2", "3"]:
        pass
    elif action == "0":
        print("欢迎再次使用【名片管理系统】")

        break
    else:
```

```
    print("输入错误，请重新输入")
```

字符串判断

```
if action in ["1", "2", "3"]:
if action == "1" or action == "2" or action == "3":
```

1. 使用 `in` 针对 **列表** 判断，避免使用 `or` 拼接复杂的逻辑条件
2. 没有使用 `int` 转换用户输入，可以避免 **一旦用户输入的不是数字**，导致程序运行出错

pass

- `pass` 就是一个空语句，不做任何事情，一般用做占位语句
- 是为了保持程序结构的完整性

无限循环

- 在开发软件时，如果 **不希望程序执行后** 立即退出
- 可以在程序中增加一个 **无限循环**
- **由用户来决定** 退出程序的时机

TODO 注释

- 在 `#` 后跟上 `TODO`，用于标记需要去做的工作

```
# TODO(作者/邮件) 显示系统菜单
```

1.3 在 `cards_tools` 中增加四个新函数

```
def show_menu():

    """显示菜单
    """
    pass

def new_card():

    """新建名片
    """
```

```
    print("-" * 50)
    print("功能：新建名片")


def show_all():

    """显示全部
    """
    print("-" * 50)
    print("功能：显示全部")


def search_card():

    """搜索名片
    """
    print("-" * 50)
    print("功能：搜索名片")
```

1.4 导入模块

- 在 `cards_main.py` 中使用 `import` 导入 `cards_tools` 模块

```
import cards_tools
```

- 修改 `while` 循环的代码如下：

```
import cards_tools

while True:

    cards_tools.show_menu()

    action = input("请选择操作功能：")

    print("您选择的操作是：%s" % action)

    # 根据用户输入决定后续的操作
    if action in ["1", "2", "3"]:
```

```
        if action == "1":
            cards_tools.new_card()

        elif action == "2":
            cards_tools.show_all()

        elif action == "3":
            cards_tools.search_card()

    elif action == "0":
        print("欢迎再次使用【名片管理系统】")

        break
    else:
        print("输入错误，请重新输入：")
```

至此：`cards_main` 中的所有代码全部开发完毕！

1.5 完成 `show_menu` 函数

```
def show_menu():

    """显示菜单
    """
    print("*" * 50)
    print("欢迎使用【菜单管理系统】V1.0")
    print("")
    print("1. 新建名片")
    print("2. 显示全部")
    print("3. 查询名片")
    print("")
    print("0. 退出系统")
    print("*" * 50)
```

2. 保存名片数据的结构

程序就是用来处理数据的，而变量就是用来存储数据的

- 使用 **字典** 记录 **每一张名片** 的详细信息

- 使用 **列表** 统一记录所有的 **名片字典**

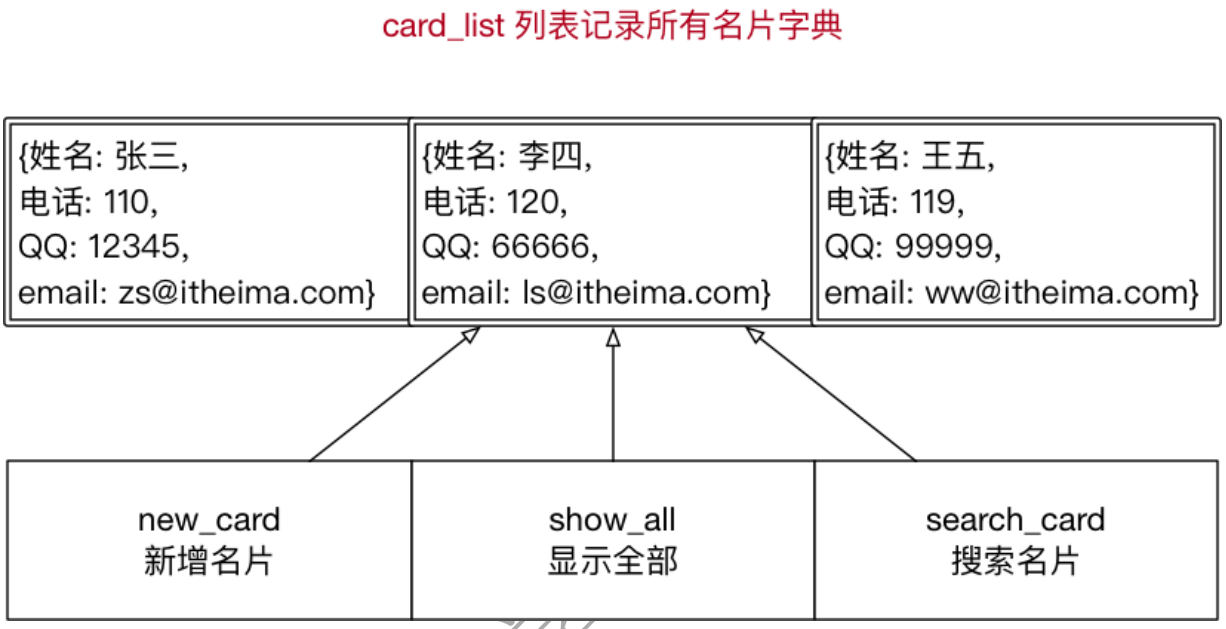

图 8-1 名片字典

定义名片列表变量

- 在 `cards_tools` 文件的顶部增加一个 **列表变量**

```
# 所有名片记录的列表
card_list = []
```

注意

1. **所有名片相关操作**，都需要使用这个列表，所以应该 **定义在程序的顶部**
2. **程序刚运行时，没有数据**，所以是 **空列表**

3. 新增名片

3.1 功能分析

1. 提示用户依次输入名片信息
2. 将名片信息保存到一个字典
3. 将字典添加到名片列表
4. 提示名片添加完成

3.2 实现 new_card 方法

- 根据步骤实现代码

```
def new_card():
```

```
    """新建名片
    """
    print("-" * 50)
    print("功能：新建名片")

    # 1. 提示用户输入名片信息
    name = input("请输入姓名：")
    phone = input("请输入电话：")
    qq = input("请输入 QQ 号码：")
    email = input("请输入邮箱：")

    # 2. 将用户信息保存到一个字典
    card_dict = {"name": name,
                 "phone": phone,
                 "qq": qq,
                 "email": email}

    # 3. 将用户字典添加到名片列表
    card_list.append(card_dict)

    print(card_list)

    # 4. 提示添加成功信息
    print("成功添加 %s 的名片" % card_dict["name"])
```

技巧：在 PyCharm 中，可以使用 SHIFT + F6 统一修改变量名

4. 显示所有名片

4.1 功能分析

- 循环遍历名片列表，顺序显示每一个字典的信息

4.2 基础代码实现

```
def show_all():

    """显示全部
```

```
    """
    print("-" * 50)
    print("功能：显示全部")

    for card_dict in card_list:

        print(card_dict)
```

- 显示效果不好！

4.3 增加标题和使用 `\t` 显示

```
def show_all():
    """显示全部
    """
    print("-" * 50)
    print("功能：显示全部")

    # 打印表头
    for name in ["姓名", "电话", "QQ", "邮箱"]:
        print(name, end="\t\t")

    print("")

    # 打印分隔线
    print("=" * 50)

    for card_dict in card_list:

        print("%s\t\t%s\t\t%s\t\t%s" % (card_dict["name"],
                                        card_dict["phone"],
                                        card_dict["qq"],
                                        card_dict["email"]))
```

4.4 增加没有名片记录判断

```
def show_all():
    """显示全部
    """
    print("-" * 50)
    print("功能：显示全部")
```

```
    # 1. 判断是否有名片记录
    if len(card_list) == 0:
        print("提示：没有任何名片记录")

        return
```

注意

- 在函数中使用 `return` 表示返回
- 如果在 `return` 后没有跟任何内容，只是表示该函数执行到此就不再执行后续的代码

5. 查询名片

5.1 功能分析

1. 提示用户要搜索的姓名
2. 根据用户输入的姓名遍历列表
3. 搜索到指定的名片后，再执行后续的操作

5.2 代码实现

- 查询功能实现

```
def search_card():

    """搜索名片
    """
    print("-" * 50)
    print("功能：搜索名片")

    # 1. 提示要搜索的姓名
    find_name = input("请输入要搜索的姓名：")

    # 2. 遍历字典
    for card_dict in card_list:

        if card_dict["name"] == find_name:

            print("姓名\t\t\t 电话\t\t\tQQ\t\t\t 邮箱")
            print("-" * 40)
```

```
                print("%s\t\t\t%s\t\t\t%s\t\t\t%s" % (
                    card_dict["name"],
                    card_dict["phone"],
                    card_dict["qq"],
                    card_dict["email"]))

                print("-" * 40)

                # TODO(小明) 针对找到的字典进行后续操作：修改/删除

                break
        else:
            print("没有找到 %s" % find_name)
```

- 增加名片操作函数：**修改/删除/返回主菜单**

```
def deal_card(find_dict):
    """操作搜索到的名片字典
    :param find_dict:找到的名片字典
    """
    print(find_dict)
    action_str = input("请选择要执行的操作 "
                       "[1] 修改 [2] 删除 [0] 返回上级菜单")
    if action == "1":
        print("修改")
    elif action == "2":
        print("删除")
```

6. 修改和删除

6.1 查询成功后删除名片

- 由于找到的字典记录已经在列表中保存
- 要删除名片记录，只需要把列表中对应的字典删除即可

```
    elif action == "2":
        card_list.remove(find_dict)
        print("删除成功")
```

6.2 修改名片

- 由于找到的字典记录已经在列表中保存
- 要修改名片记录，只需要把列表中对应的字典中每一个键值对的数据修改即可

```
if action == "1":
    find_dict["name"] = input("请输入姓名：")
    find_dict["phone"] = input("请输入电话：")
    find_dict["qq"] = input("请输入 QQ：")
    find_dict["email"] = input("请输入邮件：")

    print("%s 的名片修改成功" % find_dict["name"])
```

修改名片细化

- **如果用户在使用时，某些名片内容并不想修改**，应该如何做呢？—— 既然系统提供的 `input` 函数不能满足需求，那么就新定义一个函数 `input_card_info` 对系统的 `input` 函数进行扩展

```
def input_card_info(dict_value, tip_message):
    """输入名片信息
    :param dict_value: 字典原有值
    :param tip_message: 输入提示信息
    :return: 如果输入，返回输入内容，否则返回字典原有值
    """
    # 1. 提示用户输入内容
    result_str = input(tip_message)
    # 2. 针对用户的输入进行判断，如果用户输入了内容，直接返回结果
    if len(result_str) > 0:

        return result_str
    # 3. 如果用户没有输入内容，返回 `字典中原有的值`
    else:
        return dict_value
```

7. LINUX 上的 `Shebang` 符号(`#!`)

- `#!`这个符号叫做 `Shebang` 或者 `Sha-bang`

- Shebang 通常在 Unix 系统脚本的中 **第一行开头** 使用
- 指明 **执行这个脚本文件** 的 **解释程序**

使用 Shebang 的步骤

1. 使用 which 查询 python3 解释器所在路径

```
$ which python3
```

2. 修改要运行的 **主 python 文件**，在第一行增加以下内容

```
#! /usr/bin/python3
```

3. 修改 **主 python 文件** 的文件权限，增加执行权限

```
$ chmod +x cards_main.py
```

4. 在需要时执行程序即可

```
./cards_main.py
```

第十一章 使用 Python 编写网络爬虫

学习目标

- 爬虫环境配置
- Requests
- BeautifulSoup
- requests 与 BeautifulSoup
- 爬取新浪新闻正文
- 完整爬虫代码

1 爬取前的准备

- 打开 cmd 窗口，进入 python 安装目录
  - 下载 python，配置环境（可使用 anocanda，里面提供了很多 python 模块）
  - BeautifulSoup 的导入：pip install BeautifulSoup4
  - requests 的导入：pip install requests
  - pandas 的导入：pip install pandas
  - 下载 jupyter notebook(可选)：pip install jupyter notebook

2 requests 示例

- 下面是在 python 中使用 requests 包中 get 方法的小例子

```
#requests.get 示例
import requests
res=requests.get('http://news.sina.com.cn/china/')
res.encoding='utf-8'  #这一句是为了避免中文乱码
print(res)    #输出结果是<Response [200]>，可知 resquests.get 返回回复的数量，而不是回复的内容
print(res.text)  #因此加上“.text”才是得到网页内容
```

- 输出结果的前面一小部分显示如下。可以发现，requests.get 获取的内容是一个完整的 HTML 文档，不仅包括网页显示的信息，还包括了 html 的标签。为了将这些标签去掉，就需要使用到 BeautifulSoup4。

```
<Response [200]>
<!DOCTYPE html>
<!-- [ published at 2018-08-09 19:30:48 ] -->
<html>
<head>
<meta http-equiv="Content-type" content="text/html; charset=utf-8" />
<title>国内新闻_新闻中心_新浪网</title>
<meta name="keywords" content="国内时政,内地新闻">
<meta name="description" content="新闻中心国内频道，纵览国内时政、综述评论及图片的栏目，主要包括时政要闻、内地新闻、港澳台新闻、媒体聚焦、评论分析。">
<meta name="robots" content="noarchive">
<meta name="Baiduspider" content="noarchive">
<meta http-equiv="Cache-Control" content="no-transform">
<meta http-equiv="Cache-Control" content="no-siteapp">
<meta name="applicable-device" content="pc,mobile">
<meta name="MobileOptimized" content="width">
<meta name="HandheldFriendly" content="true">
```

- requests.get 只有当想要爬取的网页的请求方式(request method)是 GET 时才能用，可以用浏览器自带的功能进行查看。例如在 chrome 浏览器中，打开一个网页，在页面空白处 右击->检查，可以打开开发者界面。刷新页面并在 Name 属性下选中第一个（大部分都是第一个），在右侧可以看到 Resquest Method:GET，表明该网页的请求方式是 GET 方式

3 BeautifulSoup 示例

- 下面是一些示例，通过一个简单的 html 脚本来演示 BeautifulSoup 的工作。

3.1 示例：简单的 BeautifulSoup 示例

```
#简单的 BeautifulSoup 示例
from bs4 import BeautifulSoup
html_sample='\
<html> \
```

```
        <body> \
        <h1 id="title">hello world</h1> \
        <a href="#" class="link">This is link1</a> \
        <a href="# link2" class="link">This is link2</a> \
        </body> \
</html>'

soup=BeautifulSoup(html_sample,'html.parser')#指定解析器 html.parser
print(soup) #没有去掉标签
print(soup.text) #把里面的内容截取出来，而去掉标签
```

- 运行结果为:

```
<html> <body> <h1 id="title">hello world</h1> <a class="link" href="#">This is link1</a> <a class="link" href="# link2">This is link2</a> </body> </html>
hello world This is link1 This is link2
```

3.2 示例：使用 select 找出含有 h1 标签的元素

```
#使用 select 找出含有 h1 标签的元素
soup = BeautifulSoup(html_sample,'html.parser')
header = soup.select('h1')
print(header)#回传 Python 的一个 list
print(header[0])#解开这个回传的 list，打[0]时没有两边的中括号
print(header[0].text)#只获取里面的文字
```

- 运行结果为:

```
[<h1 id="title">hello world</h1>]
<h1 id="title">hello world</h1>
hello world
```

3.3 示例：使用 select 找出含有 a 标签的元素

```
#使用 select 找出含有 a 标签的元素
soup = BeautifulSoup(html_sample,'html.parser')
alink = soup.select('a')
```

```
print(alink)
for link in alink:
        print(link)
        print(link.text)
```

- 运行结果为:

```
[<a class="link" href="#">This is link1</a>, <a class="link" href="# link2">This is link2</a>]
<a class="link" href="#">This is link1</a>
This is link1
<a class="link" href="# link2">This is link2</a>
This is link2
```

3.4 示例：使用 select 找出所有 id 为 title 的元素

```
#使用 select 找出所有 id 为 title 的元素
soup = BeautifulSoup(html_sample,'html.parser')
alink = soup.select('#title')  #（id 前面需加上#）
print(alink)
```

- 运行结果为:

```
[<h1 id="title">hello world</h1>]
```

3.5 示例：取得所有 a 标签内的链接

```
#取得所有 a 标签内的链接
#使用 select 找出所有的 a tag 的 href 连接
alinks=soup.select('a')
for link in alinks:
        print(link)
        print(link['href'])#用中括号去的里面的词，因为 select 把取得大部分包装成起来
```

- 运行结果为:

```
<a class="link" href="#">This is link1</a>
#
<a class="link" href="# link2">This is link2</a>
# link2
```

3.6 示例：获取 a 标签中的不同属性值

```
#获取 a 标签中的不同属性值
a='<a href="#" qoo=123 abc=456> i am a link </a>'
soup2=BeautifulSoup(a, 'html.parser')
#print(link['href'])
print(soup2.select('a'))
print(soup2.select('a')[0])
print(soup2.select('a')[0]['href']) #最后一个中括号里面可以是'abc','qoo','href'，放入不同的属性名称，就可以取得对应的属性值
```

- 运行结果为:

```
[<a abc="456" href="#" qoo="123"> i am a link </a>]
<a abc="456" href="#" qoo="123"> i am a link </a>
#
#
```

4 将 requests 与 BeautifulSoup 结合使用的一些例子

4.1 示例：新浪新闻主页信息获取

```
#新浪新闻主页信息获取
import requests
from bs4 import BeautifulSoup
res =requests.get('http://news.sina.com.cn/china/')
res.encoding='utf-8'
soup=BeautifulSoup(res.text, 'html.parser')
    #soup=BeautifulSoup(res, 'html.parser')
    #print(soup.select('res')[0].text) 这两行是错误的方法，res 是 response 型，不能直接在 BeautifulSoup 中调用
```

```
#print(res.text)  #获取新浪新闻主页的全部信息

###仅提取日期、新闻标题、来源的全部列表
for news in soup.select('.news-item'):
    if len(news.select('h2'))>0:  #这个判断是为了避免取出的有空值，注意冒号
        h2=news.select('h2')[0].text
        time=news.select('.time')[0].text  #因为 time 属性是 class，显示所有时要加点“.time”
        a=news.select('a')[0]['href']
        print(time,h2,a)
```

- 运行结果前几条如下:

```
8 月 9 日  21:59  《新闻联播》连播五篇央视快评  多家境外媒体转发
http://news.sina.com.cn/c/2018-08-09/doc-ihhnunsq3045458.shtml
8 月 9 日  21:45  新华时评：聚焦科技高地  决胜创新未来
http://news.sina.com.cn/o/2018-08-09/doc-ihhnunsq3004687.shtml
8 月 9 日  21:40  黑龙江发生疑似羊炭疽疫情  1 人诊断为疑似病例
http://news.sina.com.cn/c/2018-08-09/doc-ihhnunsq2951111.shtml
8 月 9 日  21:23  私存反动杂志的厅官被双开  通报还有一处极罕见
http://news.sina.com.cn/o/2018-08-09/doc-ihhnunsq2890626.shtml
8 月 9 日  21:17 “绿委”批大陆水贵  遭金门全方位多角度打脸
http://news.sina.com.cn/c/2018-08-09/doc-ihhnunsq2846639.shtml
8 月 9 日  21:14  南海热带低压 10 日或登陆海南  国家防总部署防御
http://news.sina.com.cn/o/2018-08-09/doc-ihhnunsq2857167.shtml
```

4.2 示例：获取某一篇文章的标题、日期、来源、正文等内容

```
#获取某一篇文章的标题、日期、来源、正文等内容
import requests
from bs4 import BeautifulSoup
res
=requests.get('http://news.sina.com.cn/o/2018-03-16/doc-ifysiesm9100707.shtml'
)
res.encoding='utf-8'
#print(res.text)  #这里得到的是带有 html 标签的网页内容
```

```
soup=BeautifulSoup(res.text, 'html.parser')  #将 requests.get 获取得到的 res 对象变成 BeautifulSoup 对象，供后面的每个字段属性的提取

title=soup.select('.main-title')[0].text #获取标题
datesource=soup.select('.date')[0].text  #获取日期
source=soup.select('.source')[0].text  #获取来源
sourcelink=soup.select('.source')[0]['href']  #获取来源链接
article=soup.select('.article')[0].text  #获取正文内容
print(title,datesource,source,sourcelink,article)
```

- 运行结果为:

```
陈宝生：2017 年在消除大班额方面取得突破性进展  2018 年 03 月 16 日  12:06  央视
http://m.news.cctv.com/2018/03/16/ARTIRUqtoX50IsMNaaSh7dN1180316.shtml
    原标题  [微视频]陈宝生：2017 年在消除大班额方面取得突破性进展
    今天（3 月 16 日）上午，十三届全国人大一次会议新闻中心举行记者会，邀请教育部部长陈宝生就“努力让每个孩子都能享有公平而有质量的教育”等问题回答中外记者提问。
    陈宝生部长就消除城镇学校大班额现象回答了记者的提问。
    教育部部长陈宝生：2017 年在消除大班额方面，我们取得了突破性的进展。2017 年，我们有大班额 36.8 万个班，占全部班级的 10.1%，这一年我们减少了 8.2 万个大班额，和上年比下降了 18.3%，这个幅度是很大的。超大班额，现在我们全国有 8.6 万个班，占全部班级的 2.4%。去年一年，我们减少了 5.6 万个班，下降的比例是很大的，39%以上，近 40%，这是去年在消除大班额方面取得的突破性进展。
    （略）
```

4.3 示例：输出字符串型的 date 和时间型的 date

- 示例 3.3 中获取的日期的数据类型是字符串型，可以用：type(datesource)语句进行查看，输出会得到： str 。但当在实际用途中，可能需要对时间数据进行类型转换，一个简单的方法是用 datetime 包中 的 strftime 方法

```
#输出字符串型的 date 和时间型的 date
from datetime import datetime
dt=datetime.strptime(datesource,'%Y 年%m 月%d 日  %H:%M')
```

```
print(dt)
type(dt)
```

- 运行结果为:

```
2018-03-16 12:06:00
datetime.datetime
```

5 对新闻正文内容的抓取

5.1 示例：输出字符串型的 date 和时间型的 date

- 在对新闻网页的新闻文本进行提取时，通常文本会分为多个段落，也就是会有多个

  标签，例如对于下面的一个网页，新闻正文存放在 id 为 artibody 的 div 标签内，每一段分成一个 p 标签内。

- 要获取正文的全部内容的代码如下:

```
import requests
from bs4 import BeautifulSoup
res
=requests.get('http://news.sina.com.cn/gov/2017-11-02/doc-ifynmzrs6000226.shtml')
res.encoding='utf-8'

soup=BeautifulSoup(res.text, 'html.parser')
title=soup.select('#artibody')[0].text

print(title)
```

- 运行结果为:

```
中新网北京 11 月 2 日电（记者 李金磊）人社部 1 日召开第三季度新闻发布会，回应了最低工资、养老保险全国统筹、就业形势、农民工工资等一系列民生热点问题：截至 10 月底，全国共有 17 个地区调整了最低工资标准；养
```

老保险全国统筹准备明年迈出第一步；三季度末全国城镇登记失业率为金融危机以来最低点；将加大对拖欠农民工工资违法行为的查处力度。
17 地区发布 2017 年最低工资标准。
　　17 个地区已经调整最低工资标准
　　人社部新闻发言人卢爱红介绍，截至 10 月底，全国共有 17 个地区调整了最低工资标准，平均调增幅度 10.4%。全国月最低工资标准最高的是上海的 2300 元，小时最低工资标准最高的是北京的 22 元。
（略）

- 几乎是和原网页显示的一样了，段落分明且不带标签。

6 对使用了 javascript 方式的评论数的抓取

- 一般来说，新闻网页的评论数通过字段名的方式是无法直接获取的。
- 比如找到了评论数在 commentCount1M 字段内，
- 于是直接用 soup.select：

```
soup.select('#commentCount1M')
```

- 但发现得到的结果并没有

```
[<span id="commentCount1M"></span>]
```

- 这是因为，评论数往往不是一个静态的数字，是通过 javascript 方式的得到的，它会根据评论数量的增加而改变的。所以正确的方式应该是找到它所使用的 js。
- 操作方式如下：
- 1、在网页的 js 文件下查找含有评论数都文件，比如我抓取的网页的评论数是 134，那么久寻找含有这个数字的文件，再看是不是我们要的评论数。
- 可以通过 preview 视图进行快速查看。找到之后，右侧在 Hearder 视图中获取这个文件的 reuqest url。

```
import requests
res=requests.get('http://comment5.news.sina.com.cn/page/info?version=1&format=js&channel=gn&newsid=comos-fynmzrs6000226&group=&compress=0&ie=utf-8&oe=utf-8&page=1&page_size=20&')
print(res.text)
```

- 执行得到的结果是很大一片的代码，有点像 json 的脚本。
- 2、再通过 json 进行打开：

```
import json
jd=json.loads(comments.text.strip('var data='))  #转化成字典
```

- 此时就把内容加载进 jd 中。想要查看可以直接打印 jd 并运行：

```
print(jd)
```

- 运行结果会显示每一条评论的相关信息：

```
{'result':
   {'status': {'msg': '', 'code': 0},
   'count': {'qreply': 96, 'total': 176, 'show': 43},
   'replydict': {},
   'language': 'ch',
   'encoding': 'utf-8',
   'top': [],
   （略）
```

- 这说明我们成功的获取了新闻的评论。
- 3、但我们想要的是总评论数啊！在运行结果中，
- 发现 176 位于 result 下的 count 下的 total 里面，于是：

```
print(jd['result']['count']['total'])
```

- 得到结果

```
176
```

- 获取成功。

7 获取网页 url 的 id

7.1 方法一：通过使用 split（）和 strip（）函数

- 网页的 id 是网页的标识，一个 id 对应了一个网页，当我们抓取的网页数量较多时，可能需要获取每个网页对应的 id 来对其进行更方便的管理和使用。
- 每个网页的 id 在它的 url 里面很容易能找到，例如：
- http://news.sina.com.cn/gov/2017-11-02/doc-ifynmzrs6000226.shtml 中，可以知道“fynmzrs6000226”这一段就是该网页所对应的网页 id。
- 获取 id 的代码如下：

```
newsurl='http://news.sina.com.cn/gov/2017-11-02/doc-ifynmzrs6000226.shtml'
newsid=newsurl.split('/')[-1].rstrip('.shtml').lstrip('doc-i')
print(newsid)
```

- 运行结果为：

```
fynmzrs6000226
```

7.2 方法二：通过使用正则表达式

```
#获取网页 id（正则表达式法）
import re
m=re.search('doc-i(.+).shtml',newsurl)
print(m) #查看匹配结果，match 部分是匹配到的部分
print(m.group(0)) #匹配到的部分
print(m.group(1)) #匹配到的“(.+)”部分
```

- 运行结果为：

```
<_sre.SRE_Match object; span=(39, 64), match='doc-ifynmzrs6000226.shtml'>
doc-ifynmzrs6000226.shtml
fynmzrs6000226fynmzrs6000226
```

8 完整代码（以获取新浪新闻为例）

```
# 获取新闻的标题，内容，时间和评论数
import requests
from bs4 import BeautifulSoup
from datetime import datetime
import re
import json
import pandas

def getNewsdetial(newsurl):
        res = requests.get(newsurl)
        res.encoding = 'utf-8'
        soup = BeautifulSoup(res.text,'html.parser')
        newsTitle = soup.select('.page-header h1')[0].text.strip()
        nt = datetime.strptime(soup.select('.time-source')[0].contents[0].strip(),'%Y
年%m 月%d 日%H:%M')
        newsTime = datetime.strftime(nt,'%Y-%m-%d %H:%M')
        newsArticle = getnewsArticle(soup.select('.article p'))
        newsAuthor = newsArticle[-1]
        return newsTitle,newsTime,newsArticle,newsAuthor
def getnewsArticle(news):
        newsArticle = []
        for p in news:
                newsArticle.append(p.text.strip())
        return newsArticle

# 获取评论数量

def getCommentCount(newsurl):
        m = re.search('doc-i(.+).shtml',newsurl)
        newsid = m.group(1)
        commenturl =
'http://comment5.news.sina.com.cn/page/info?version=1&format=js&channel=gn
&newsid=comos-{}&group=&compress=0&ie=utf-8&oe=utf-8&page=1&page_
size=20'
        comment = requests.get(commenturl.format(newsid))      #将要修改的地方换
成大括号，并用 format 将 newsid 放入大括号的位置
        jd = json.loads(comment.text.lstrip('var data='))
        return jd['result']['count']['total']
```

```
def getNewsLinkUrl():
#       得到异步载入的新闻地址（即获得所有分页新闻地址）
        urlFormat =
'http://api.roll.news.sina.com.cn/zt_list?channel=news&cat_1=gnxw&cat_2==gd
xw1||=gatxw||=zs-pl||=mtjj&level==1||=2&show_ext=1&show_all=1&show_num
=22&tag=1&format=json&page={}&callback=newsloadercallback&_=1501000
415111'
        url = []
        for i in range(1,10):
                res = requests.get(urlFormat.format(i))
                jd = json.loads(res.text.lstrip('    newsloadercallback(').rstrip(');'))
                url.extend(getUrl(jd))          #entend 和 append 的区别
        return url

def getUrl(jd):
#       获取每一分页的新闻地址
        url = []
        for i in jd['result']['data']:
                url.append(i['url'])
        return url

# 取得新闻时间，编辑，内容，标题，评论数量并整合在 total_2 中
def getNewsDetial():
        title_all = []
        author_all = []
        commentCount_all = []
        article_all = []
        time_all = []
        url_all = getNewsLinkUrl()
        for url in url_all:
                title_all.append(getNewsdetial(url)[0])
                time_all.append(getNewsdetial(url)[1])
                article_all.append(getNewsdetial(url)[2])
                author_all.append(getNewsdetial(url)[3])
                commentCount_all.append(getCommentCount(url))
        total_2 =
{'a_title':title_all,'b_article':article_all,'c_commentCount':commentCount_all,'d_ti
me':time_all,'e_editor':author_all}
        return total_2
```

```
# (  运行起始点  )用 pandas 模块处理数据并转化为 excel 文档

df = pandas.DataFrame(getNewsDetial())
df.to_excel('news2.xlsx')
```